Simulation of a morphological image processor using VHDL - Part I: Mathematical Components

Wei-chun Chen

Follow this and additional works at: https://scholarworks.rit.edu/theses

Recommended Citation

This Thesis is brought to you for free and open access by RIT Scholar Works. It has been accepted for inclusion in Theses by an authorized administrator of RIT Scholar Works. For more information, please contact ritscholarworks@rit.edu.
Simulation of A Morphological Image Processor

Using VHDL

– Part I: Mathematical Components

Wei-chun Chen

A Thesis Submitted in Partial Fulfilment Of the Requirements for the degree of

Master of Science

in

Computer Engineering

Approved by:  Professor  

___________________________  
George A. Brown (Thesis Advisor)

Professor  

___________________________  
Tony H. Chang, Ph. D.

Professor  

___________________________  
Roy S. Czernikowski, Ph. D.

Department of Computer Engineering

College of Engineering

Rochester Institute of Technology

Rochester, New York

February 1993
Title of thesis:

Simulation of A Morphological Image Processor Using VHDL
– Part I: Mathematical Components

hereby grant permission to the Wallace Memorial Library and Computer Engineering Department of RIT to reproduce my thesis in whole or in part. Any reproduction will not be for commercial use or profit.

Date: March 1, 1993
Acknowledgments

We wish to express our grateful appreciation to the many people who helped make this thesis a reality. We are particularly indebted to Professor George A. Brown for his invaluable advice and encouragement throughout this entire project. Special thanks are due to Jens Rodenberg who provided enormous information about the MIP system design. We are also thankful to Professor Roy S. Czernikowski and Tony H. Chang who painstakingly read the entire manuscript and made valuable suggestions. Our sincere appreciation is extended to Jeff Hanzlik, Chris Insalaco, Shishir Ghate, and Larry Robin for their previous effort on the MIP project.
Abstract

Very high speed integrated circuit Hardware Description Language (VHDL) is utilized in this project to model a Morphological Image Processor (MIP) Array. Both behavioral and structural models have been established at the system level, and the simulation results from both models are consistent with each other. The successful implementation of the models accomplishes our original goal to document the MIP with VHDL. It is observed from the project that VHDL is a powerful language. It is flexible since it can be used to model any level of a system independent of the technology.
## Glossary

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>Arithmetic/Logic Unit</td>
</tr>
<tr>
<td>ASCII</td>
<td>American Standard Code for Information Interchange</td>
</tr>
<tr>
<td>BLM</td>
<td>Behavioral Language Model</td>
</tr>
<tr>
<td>EISA</td>
<td>Extended Industrial Standard Architecture</td>
</tr>
<tr>
<td>FIFO</td>
<td>First-In-First-Out</td>
</tr>
<tr>
<td>FP</td>
<td>Function Processing</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field Programmable Gate Array</td>
</tr>
<tr>
<td>FSP</td>
<td>Function Set Processing</td>
</tr>
<tr>
<td>IMG</td>
<td>IMaGe file format used for frame grabber</td>
</tr>
<tr>
<td>MAP</td>
<td>Morphological Array Processor</td>
</tr>
<tr>
<td>MIP</td>
<td>Morphological Image Processor</td>
</tr>
<tr>
<td>MU</td>
<td>Morphological Unit</td>
</tr>
<tr>
<td>RTL</td>
<td>Register Transfer Level</td>
</tr>
<tr>
<td>SP</td>
<td>Set Processing</td>
</tr>
<tr>
<td>TIFF</td>
<td>Tag-based Image File Format for storing and interchanging raster images.</td>
</tr>
<tr>
<td>VHDL</td>
<td>VHSIC Hardware Description Language</td>
</tr>
<tr>
<td>VHSIC</td>
<td>Very High Speed Integrated Circuit</td>
</tr>
<tr>
<td>VLSI</td>
<td>Very Large Scale Integrated circuit</td>
</tr>
</tbody>
</table>
# Contents

1 Introduction ........................................... 1

2 Morphology Theory ...................................... 3
   2.1 Digital Images ..................................... 3
   2.2 Basic Morphological Operations ................. 4
   2.3 Extended Morphological Operations .......... 6
   2.4 Implementation .................................. 6
   2.5 Examples ......................................... 8
      2.5.1 Example 1: .................................. 8
      2.5.2 Example 2: .................................. 9

3 VHDL ...................................................... 12
   3.1 Overview of Top-down Design .................. 12
   3.2 Entity and Architecture ......................... 14
   3.3 Signal vs. Variable ............................. 15

4 System Description ..................................... 18
   4.1 Data Path ......................................... 18
   4.2 Arithmetic Functional Blocks ................. 20
      4.2.1 ALUs ....................................... 20
      4.2.2 MU and Volume Adder ...................... 22
   4.3 Control and Status Registers .................. 23
      4.3.1 Volume Adder Registers ..................... 23
      4.3.2 ALUs ....................................... 23
      4.3.3 Master Controller ........................... 24
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.4.1 Ports</td>
<td>86</td>
</tr>
<tr>
<td>6.4.2 Processes</td>
<td>86</td>
</tr>
<tr>
<td>7 Memory Units</td>
<td>90</td>
</tr>
<tr>
<td>7.1 Memory</td>
<td>90</td>
</tr>
<tr>
<td>7.1.1 Ports</td>
<td>90</td>
</tr>
<tr>
<td>7.1.2 Process</td>
<td>90</td>
</tr>
<tr>
<td>7.2 Buffer</td>
<td>91</td>
</tr>
<tr>
<td>7.2.1 Ports</td>
<td>91</td>
</tr>
<tr>
<td>7.2.2 Process</td>
<td>92</td>
</tr>
<tr>
<td>7.2.3 Further Implementation</td>
<td>92</td>
</tr>
<tr>
<td>7.1.1 Ports</td>
<td>90</td>
</tr>
<tr>
<td>7.1.2 Process</td>
<td>90</td>
</tr>
<tr>
<td>7.2 Buffer</td>
<td>91</td>
</tr>
<tr>
<td>7.2.1 Ports</td>
<td>91</td>
</tr>
<tr>
<td>7.2.2 Process</td>
<td>92</td>
</tr>
<tr>
<td>7.2.3 Further Implementation</td>
<td>92</td>
</tr>
<tr>
<td>8 Conclusion</td>
<td>94</td>
</tr>
<tr>
<td>A Bus Interface</td>
<td>97</td>
</tr>
<tr>
<td>A.1 Input and Output Signals</td>
<td>97</td>
</tr>
<tr>
<td>A.2 VHDL Model of the Bus Interface</td>
<td>99</td>
</tr>
<tr>
<td>B Controller</td>
<td>106</td>
</tr>
<tr>
<td>B.1 Master Controller</td>
<td>106</td>
</tr>
<tr>
<td>B.1.1 Inputs and Outputs</td>
<td>106</td>
</tr>
<tr>
<td>B.1.2 VLSI Version vs. FPGA Version</td>
<td>109</td>
</tr>
<tr>
<td>B.1.3 VHDL Model of the Master Controller</td>
<td>109</td>
</tr>
<tr>
<td>B.2 Memory Controller</td>
<td>124</td>
</tr>
<tr>
<td>B.2.1 Inputs and Outputs</td>
<td>124</td>
</tr>
<tr>
<td>B.2.2 VHDL Model of the Memory Controller</td>
<td>124</td>
</tr>
<tr>
<td>C Utilities</td>
<td>128</td>
</tr>
<tr>
<td>C.1 XEROX 7650 Scanner</td>
<td>128</td>
</tr>
<tr>
<td>C.2 TIFF to IMG</td>
<td>128</td>
</tr>
<tr>
<td>C.3 IMG to ASCII / ASCII to IMG and Display an IMG Image</td>
<td>129</td>
</tr>
<tr>
<td>C.4 ASCII to PS</td>
<td>129</td>
</tr>
<tr>
<td>C.5 Display a PostScript Image on PC</td>
<td>129</td>
</tr>
<tr>
<td>C.6 Connect PC with Apollo Workstations - DPCI</td>
<td>129</td>
</tr>
</tbody>
</table>
C.7  Print out a PostScript on LaserJet on Apollo  . . . . . . . . . . . . . .  129
List of Figures

3.1 Top-down Design Process ............................................. 13

4.1 MIP Data Path ......................................................... 19

4.2 Signal Timing of Reset Bus Cycle ................................. 29

4.3 Signal Timing of Register R/W Bus Cycle ...................... 30

4.4 Signal Timing of Memory R/W Bus Cycle ........................ 31

4.5 MIP Process flow ..................................................... 33

5.1 MIP Hierarchy ......................................................... 37

5.2 MIP Behavioral Model with PC BUS .............................. 40

5.3 MIP Structural Model with PC BUS .............................. 53

5.4 MIP Timing Chart ..................................................... 55

5.5 MIP Timing Chart (continued) ..................................... 56

5.6 A Partial 32 × 32 image in ASCII format ....................... 60

5.7 Resultant Image, top-left: original image, top-right: first erosion, bottom right: second erosion, bottom left: third erosion .............. 61

5.8 Original 8-bit grey scale image .................................. 63

5.9 Resultant image of a complete 512 × 512 image after erosion ................ 64

5.10 Resultant image of two 512 × 256 images after erosion .......... 65

6.1 The Blanking Sequence of the MAP ............................... 76

A.1 Schematic of BUS Interface ......................................... 100

B.1 Schematic of Master Controller in the VLSI version ............ 110

B.2 Schematic of Master Controller in the FPGA version .......... 111

B.3 Stages of Master Controller ......................................... 113
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Threshold Representation of a Discrete, Quantized Signal</td>
<td>8</td>
</tr>
<tr>
<td>2.2</td>
<td>FSP Dilation, Erosion, Opening, and Closing</td>
<td>9</td>
</tr>
<tr>
<td>2.3</td>
<td>$f(u)$ and $g(v)$</td>
<td>9</td>
</tr>
<tr>
<td>2.4</td>
<td>illustration of a grey scale dilation</td>
<td>10</td>
</tr>
<tr>
<td>2.5</td>
<td>illustration of a grey scale erosion</td>
<td>11</td>
</tr>
<tr>
<td>4.1</td>
<td>ALUs’ Operations</td>
<td>22</td>
</tr>
<tr>
<td>4.2</td>
<td>The Map of Registers in the MIP</td>
<td>23</td>
</tr>
<tr>
<td>4.3</td>
<td>Volume Adder Registers</td>
<td>24</td>
</tr>
<tr>
<td>4.4</td>
<td>ALU Registers</td>
<td>24</td>
</tr>
<tr>
<td>4.5</td>
<td>Start Register</td>
<td>24</td>
</tr>
<tr>
<td>4.6</td>
<td>Control and Status Registers</td>
<td>25</td>
</tr>
<tr>
<td>4.7</td>
<td>Memory Select</td>
<td>26</td>
</tr>
<tr>
<td>4.8</td>
<td>Memory to Local Bus Connection</td>
<td>26</td>
</tr>
<tr>
<td>4.9</td>
<td>On-board Memory Segments</td>
<td>26</td>
</tr>
<tr>
<td>4.10</td>
<td>PC Address Control</td>
<td>27</td>
</tr>
<tr>
<td>4.11</td>
<td>System Commands</td>
<td>32</td>
</tr>
<tr>
<td>5.1</td>
<td>MIP Entity</td>
<td>39</td>
</tr>
<tr>
<td>5.2</td>
<td>The Window Array</td>
<td>62</td>
</tr>
<tr>
<td>7.1</td>
<td>Tri-state Signal Resolving Table</td>
<td>92</td>
</tr>
<tr>
<td>A.1</td>
<td>I/O Address</td>
<td>98</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

The aim of this project was to model a Morphological Image Processor (MIP) with VHDL, a hardware description language. A MIP can be used to analyze an image based on a predefined geometric shape by applying set operations on the image. The total project was accomplished by the joint efforts of Wei-chun Chen and Hao Chen due to its complexity and size. Two separate thesis topics entitled Mathematical Components and Control Mechanism have resulted from the project to create a VHDL model of the Morphological Image Processor.

In order to understand the operation of the MIP, the study of the MIP theory is first presented in Chapter 2 (by Wei-chun Chen), in which digital images are defined, and morphological operations are explained. Chapter 3 (by Hao Chen) describes the important concepts in VHDL, and the role of VHDL in the top-down design. In Chapter 4 (by Wei-chun and Hao), the MIP system is described in detail. The data path between different buses is explained, the functionality of the arithmetic blocks is discussed, and the control/status registers are presented. In addition, the interface between the MIP and the host computer is illustrated, and the procedures for operating the MIP are given. These chapters are shared by Wei-chun Chen and Hao Chen for readers to better understand the entire project. The aim of this project is to model a Morphological Image Processor (MIP) with VHDL, a hardware description language. A MIP can be used to analyze an image based on a predefined geometric shape by applying set operations on the image.

The modeling of the MIP in VHDL is discussed from Chapter 5 to Chapter 9. Based on the system description in Chapter 4, the MIP is partitioned in Chapter 5 (by Wei-chun and
Hao) into four different functional blocks: I/O Unit, Control Units, Arithmetic Units, and Memory Units. Each functional block consists of one or more physical blocks. Each physical block has its corresponding VHDL model. The behavioral and structural models of the MIP are discussed in Chapter 5 as well. Chapter 6 visits the Arithmetic Units: Arithmetic/Logic Units (ALU1 and ALU2), Morphology Unit (MU), and Volume Adder. The MU is further decomposed into First-In-First-Out (FIFO) and Morphological Array Processor (MAP). At last, the Memory and the Buffer in Memory Units are discussed in Chapter 7.

Appendix A and appendix B are written by Hao Chen to describe the control mechanism of the MIP. In appendix A, the Bus Interface in I/O unit is described. Appendix B deals with the Controller Units, which consist of the Master Controller and the Memory Controller.

The original architecture and the FPGA version of the MIP were designed by Jens Rodenberg and Jeff Hanzlik. The first VLSI version of the MAP was designed by Larry Rubin. The VLSI version of the ALU1, the ALU2, the Volume Adder, and the revised VLSI version of the MAP were designed by Shishir Ghate. The VLSI version of the Master Controller and Memory Controller were designed by Chris Insalaco. All of the BLM models for the components in the MIP were written by Jeff Hanzlik. The VHDL models of the units are based on the BLM models, as well as the schematics from various versions mentioned above.
Chapter 2

Morphology Theory

According to the Webster Electronic Dictionary, the word *morphology* refers to the study of form and structure. In image processing, morphology was first used by G. Matheron as a methodology which analyzes an image based on a predefined geometric shape by applying set operations on the image. The image operations of mathematical morphology, known as morphological filters, are more suitable for shape analysis than linear filters [3].

The morphology theory can be applied on either binary or grey-scale images. The definitions of an image and morphology operations will be presented in Section 2.1. The hardware implementation of the operations will be explained in Section 2.4.

2.1 Digital Images

A digital image is normally created by the process of sampling a continuous image. The image can be represented as a function whose domain is a subset of a discrete space and whose range is a subset of integers. An element in the domain represents the coordinate of a pixel while an element in the range is the signal strength of a pixel. An image consisting of monochromatic pixels is referred as a grey scale image.

A grey scale image can be converted to a binary image by applying a thresholding processing. The thresholding process defines the corresponding pixel value as "1" when the grey-scale pixel value is larger or equal to the threshold value; otherwise, the pixel value is "0". The binary image can be represented by a set of coordinates for those pixels with value "1".

In our discussion, the discrete space is limited to two dimensions for convenience.
A binary image is defined as a set, $X$:

$$X = \{ x | x \text{ is the coordinate of the image, } x \in Z^2 \}. \quad (2.1)$$

$Z$ is the set of integers. $Z^2$ is a two dimensional discrete space.

A grey-scale image is defined as a function, $f$:

$$f : E \rightarrow F, \begin{cases} f(x) = \{ y | y \in F \} & \text{if } x \in E \\ -\infty & \text{otherwise} \end{cases} \quad (2.2)$$

$F$ is the range of the function and $E$ is the domain of the function. $F \subset Z$, $E \subset Z^2$.

The thresholding process is accomplished by obtaining thresholding sets for a grey-scale image. A thresholding set is defined by

$$T_a(f) = \{ x | f(x) \geq a, a \in Z, x \in E \}, -\infty < a < \infty. \quad (2.3)$$

in which $a$ is a threshold value. It is clear by comparing equations 2.1 and 2.3 that a thresholding set is a binary image. For an 8-bit grey-scale image, there are 256 threshold values from 0 to 255. After the thresholding process, the grey-scale image is decomposed into 256 binary images.

The decomposed grey-scale image can be reconstructed by the operation:

$$f(x) = \max\{ a : x \in T_a(f) \}, \forall x. \quad (2.4)$$

2.2 Basic Morphological Operations

The morphological operations are based on a structuring element to analyze an input image. Any small sized image with a simple shape can be used as a structuring element. There are three categories of processing classified by the input images and the structuring elements. We will define two basic operations in each category: dilation and erosion.

The first category is set processing (SP) with a binary input image and a structuring element. Let $X \pm b = \{ x \pm b : x \in X \}$ denote the vector translate of $X$ by $\pm b$. The dilation
of a binary image $X$ by a binary structuring element $B$ is defined as:

$$X \oplus B = \bigcup_{b \in B} (X + b) = \{x + b \mid \forall x \in X \land \forall b \in B\} \quad (2.5)$$

Dilating an image by a structuring element $B$ has the effect of "expanding" the image in a manner determined by $B$.

The erosion of a binary image $X$ by a binary structuring element $B$ is defined as:

$$X \ominus B = \bigcap_{b \in B} (X - b) = \{x \mid x \in X \land (B + x) \subseteq X\}. \quad (2.6)$$

Eroding an image by a structuring element $B$ has the effect of "shrinking" the image in a manner determined by $B$.

The second category is function set processing (FSP) with a grey-scale input image and a binary structuring element. An important property of the FSP operations is that they commute with thresholding. That is, let $\phi$ denote a FSP operation and let $\Phi$ denote its respective SP operation. Then, we say that $\phi$ commutes with thresholding iff

$$\Phi[T_a(f)] = T_a[\phi(f)], \forall t \in Z. \quad (2.7)$$

The dilation of a grey-scale image $f$ by a binary structuring element $B$ is defined as:

$$(f \oplus B)(x) = \max_{y \in B} \{f(x - y), x - y \in E\}. \quad (2.8)$$

The erosion of a grey-scale image $f$ by a binary structuring element $B$ is defined as:

$$(f \ominus B)(x) = \min_{y \in B} \{f(x + y), x + y \in E\}. \quad (2.9)$$

As seen from equation 2.7, the FSP operation is equivalent to 256 SP operations for a 8-bit grey-scale image. Therefore, the FSP operation has received more attention in mathematical morphology research.

The third category is function processing (FP) with a grey-scale input image and a grey-scale structuring element. Let $g$ be a function whose domain is $I \subseteq \mathbb{Z}^2$ and whose range is $J \subseteq \mathbb{Z}$. The dilation of a grey-scale image $f$ by a grey-scale structuring element $g$
is defined as:

\[ (f \oplus g)(x) = \max\{f(y) + g(x - y), x \in E, \forall (x - y) \in I\}. \tag{2.10} \]

The erosion of a grey-scale image \( f \) by a grey-scale structuring element \( g \) is defined as:

\[ (f \ominus g)(x) = \min\{f(y) - g(y - x), x \in E, \forall (y - x) \in I\}. \tag{2.11} \]

It should be realized that the FP operations do not commute with SP operations.

### 2.3 Extended Morphological Operations

The utilization of erosion and dilation can be extended to opening and closing. Any set of the erosion and the dilation operations defined above can be used in an opening or a closing operation.

The opening operation can be defined as:

\[ (X \circ B) = (X \ominus B) \oplus B. \tag{2.12} \]

The closing operation can be defined as:

\[ (X \bullet B) = (X \oplus B) \ominus B. \tag{2.13} \]

### 2.4 Implementation

Our goal is to design and simulate a grey-scale morphological processing system. In the system, images and structuring elements will be grey-scale images. A structuring element is referred as a mask in future discussions. In equation 2.2 we have shown that the number \(-\infty\) is used to represent an undefined pixel value; therefore, a 9-bit 2's complement code is used instead of a 8-bit unsigned binary code. The number \(-256\) is used to represent the \(-\infty\) value. Even though the original image from a sampling does not have negative valued pixels, the negative value for a pixel can occur during computation. The maximum image frame size is fixed to 512 by 512 (or 1024 by 1024) pixels for our system while the mask frame size is 7 by 7 pixels.
The essential operations in the MIP system are erosion and dilation since the opening or closing operations can be created using erosion and dilation. We will first explain the implementation of FP operations, then apply the result to FSP operations.

The definitions of FP operations have been shown in equation 2.10 and 2.11. The computing procedures for dilation as shown by equation 2.10 can be described by the following pseudo code:

```plaintext
for (output_col_index=0; output_col_index<512; output_col_index++)
for (output_row_index=0; output_row_index<512; output_row_index++)
{
  for (mask_col_index=0; mask_col_index<7; mask_col_index++)
  for (mask_row_index=0; mask_row_index<7; mask_row_index++)
  {
    input_row_index := output_row_index - mask_row_index + (mask_size-1)/2;
    input_col_index := output_col_index - mask_col_index + (mask_size-1)/2;
    OUTPUT_IAMGE(output_col_index, output_row_index) :=
      max( map_add(INPUT_IAMGE(input_col_index, input_row_index),
                    MASK(mask_col_index, mask_row_index )));
  }
}
```

The procedure can be visualized with the following steps:

1. rotate the mask by 180 degrees;
2. align the target pixel (starting from column 0, row 0 of the image) with the center of the mask;
3. add the corresponding pixels between the mask and the part of the image overlapped with the mask;
4. choose the maximum value of the summations as the value of the target pixel;
5. slide the mask to the next target pixel and repeat the previous steps until the whole image is processed.

By comparing equation 2.11 with 2.10, the erosion operation can be computed by negating the mask pixel value but not rotating mask. However, the target value is now the minimum value of the summations.

The negating and rotating mask procedures for erosion and dilation can be easily accomplished by a microcomputer such as an IBM PC without enlarging the hardware. There-
fore, the system has been partitioned to compute the summations and search the maximum/minimum by hardware, and to rotate and negate the mask by software.

The reader should be aware that if a \(-\infty\) is used in a mask, the negated value is \(+\infty\) which is not defined in our 9 bit 2's complement coded system. This technical problem can be solved by using the maximum value 255 instead of \(+\infty\). The trade off is that the actual value of 255 is indistinguishable from \(+\infty\).

The FSP operation can be applied on this system by using a proper mask. In the mask, 0 is used for the image and \(-\infty\) for the background.

2.5 Examples

The following two examples are modified based on the works of Haralick [2] and Morgos [4].

2.5.1 Example 1:

We will illustrate through an example in table 2.1 the procedure of thresholding a 1-D grey scale image as well as the result of FSP operations on the image. The first row shows the coordinate of the pixel, while the second row shows the value of the pixel. The resultant thresholding sets are shown from row 3 to row 6 in which '●' shows that the corresponding coordinate is an element in the set. The equivalent representation of the thresholding sets is the binary images which are shown from row 7 to row 10. The original grey-scale function can be reconstructed by restoring the maximum value showing on the thresholding sets to the correspondent coordinate.

<p>| | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1:</td>
<td>x</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>2:</td>
<td>f(x)</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3:</td>
<td>T_3(f)</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4:</td>
<td>T_2(f)</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5:</td>
<td>T_1(f)</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td></td>
<td>●</td>
<td>●</td>
<td>●</td>
<td></td>
</tr>
<tr>
<td>6:</td>
<td>T_0(f)</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>7:</td>
<td>f_3(x)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8:</td>
<td>f_2(x)</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9:</td>
<td>f_1(x)</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>10:</td>
<td>f_0(x)</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 2.1: Threshold Representation of a Discrete, Quantized Signal
The \( f(x) \) is processed by a structuring set \( B = \{-1, 0, 1\} \) for dilation, erosion, opening, and closing as shown in table 2.2.

Each result of table 2.2 can be used to threshold and generate another resultant binary image sets. As the equation 2.7 shown, these resultant sets should be identical to the resultant sets using the row seven to ten of table 2.1 as input and applied the set \( B \) for the same FSP operations.

<table>
<thead>
<tr>
<th>( x )</th>
<th>-1</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f(x) )</td>
<td>(-\infty)</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>(-\infty)</td>
</tr>
<tr>
<td>( f(x) \oplus B )</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>( f(x) \ominus B )</td>
<td>(-\infty)</td>
<td>(-\infty)</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>(-\infty)</td>
<td>(-\infty)</td>
<td>(-\infty)</td>
</tr>
<tr>
<td>( f(x) \oslash B )</td>
<td>(-\infty)</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>(-\infty)</td>
</tr>
<tr>
<td>( f(x) \bullet B )</td>
<td>(-\infty)</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>(-\infty)</td>
<td>(-\infty)</td>
</tr>
</tbody>
</table>

Table 2.2: FSP Dilation, Erosion, Opening, and Closing

2.5.2 Example 2:

The \( f(u) \) and \( g(v) \) in table 2.3 are the coordinate and pixel value of two grey-scale images. The table 2.4 and 2.5 are the computing results according to the equation 2.10 and 2.11.

<table>
<thead>
<tr>
<th>( u )</th>
<th>15</th>
<th>16</th>
<th>17</th>
<th>18</th>
<th>19</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f(u) )</td>
<td>4</td>
<td>7</td>
<td>-5</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>( v )</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( g(v) )</td>
<td>1</td>
<td>17</td>
<td>-3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2.3: \( f(u) \) and \( g(v) \)
<table>
<thead>
<tr>
<th>$x = 15$</th>
<th>$x = 16$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x - y$</td>
<td>0</td>
</tr>
<tr>
<td>$y$</td>
<td>15</td>
</tr>
<tr>
<td>$f(y)$</td>
<td>4</td>
</tr>
<tr>
<td>$g(x - y)$</td>
<td>1</td>
</tr>
<tr>
<td>$f + g$</td>
<td>5</td>
</tr>
<tr>
<td>$(f \oplus g)(15) = 5$</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$x = 17$</th>
<th>$x = 18$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x - y$</td>
<td>0</td>
</tr>
<tr>
<td>$y$</td>
<td>17</td>
</tr>
<tr>
<td>$f(y)$</td>
<td>-5</td>
</tr>
<tr>
<td>$g(x - y)$</td>
<td>1</td>
</tr>
<tr>
<td>$f + g$</td>
<td>-4</td>
</tr>
<tr>
<td>$(f \oplus g)(17) = 24$</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$x = 19$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x - y$</td>
</tr>
<tr>
<td>$y$</td>
</tr>
<tr>
<td>$f(y)$</td>
</tr>
<tr>
<td>$g(x - y)$</td>
</tr>
<tr>
<td>$f + g$</td>
</tr>
<tr>
<td>$(f \oplus g)(19) = 23$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$x = 15$</th>
<th>$x = 16$</th>
<th>$x = 17$</th>
<th>$x = 18$</th>
<th>$x = 19$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(f \oplus g)(x)$</td>
<td>5</td>
<td>21</td>
<td>24</td>
<td>12</td>
</tr>
</tbody>
</table>

Table 2.4: illustration of a grey scale dilation
Table 2.5: illustration of a grey scale erosion
Chapter 3

VHDL

VHDL is an acronym for VHSIC Hardware Description Language. It is an industry standard language used to describe a digital system from abstract level to concrete level. One of the important features of the language is that it has constructs that enables a designer to express the concurrent or sequential behavior of a digital system with or without timing. The advantage of using VHDL is clearly illustrated in top-down design methodology. We will discuss the top-down design and the important role of VHDL in the design before moving on to the topics in VHDL.

3.1 Overview of Top-down Design

In design methodologies, top-down design is very significant since it reduces the design cycle, increases the design flexibility, and improves the design quality and productivity.

The top-down design methodology is a process that begins with establishing an architecture and defining a behavioral logic description using a high-level hardware description such as VHDL. This could be at a system or block diagram level, or register transfer level (RTL), but it begins at some point above gate level. [5] Figure 3.1 shows the top-down design process.

The behavioral model is evaluated against its architecture specifications using realistic test inputs via simulation to determine the correctness. Once the model is verified, the designer can further decompose the model into sub-models as desired, and evaluate the sub-models accordingly. Simulating at successive levels of the design process and making corrections minimizes or avoids flaws appearing at the most costly stage: when the design is
Figure 3.1: Top-down Design Process

1. Define Design Specification
2. Partition by Functionality
3. Develop VHDL Behavioral Models
4. Develop VHDL Structural Models
5. Functional Simulation
6. Technology Selection
7. Synthesis/ Optimization
8. Functional Verification
9. Timing Verification
10. Fault Analysis and Test
11. Layout and Detailed Analysis
12. Manufacturing
complete. In addition, simulation using VHDL allows changes to be made relatively easier and in much less time. Traditional gate-level simulation discourages trying alternative approaches since schematic capture, logic simulation, and timing analysis require a great deal of time and computing resources.

The next stage in the process is to select a technology for the design. This is another advantage of VHDL since it offers verifications independent of the technology. After the technology is selected, the VHDL models can be synthesized. Logic synthesis provides a link between VHDL and a netlist. This feature is extremely important for a large system design since it would otherwise be very difficult to keep track of a full gate-level description using schematic capture. The gate-level representation obtained after logic synthesis is then simulated and evaluated against the simulation results from the VHDL model. Next, this representation is verified through a combination of functional verification, timing analysis, and fault simulation. The circuit can be further optimized after functional verification or timing analysis to improve the performance of the design. Layout is then produced, and more detailed information is provided for additional timing adjustment and verifications. This process is iterated until the design is ready for manufacture.

After illustrating the importance of VHDL in top-down design, we will discuss some fundamental concepts in VHDL in the following sections.

### 3.2 Entity and Architecture

An entity is an abstraction of the actual hardware device. The *ENTITY* declaration specifies the name of the entity being modeled and the external interfaces of the modeled device. An entity can include other entities, and it can also be included in another entity. Therefore, VHDL supports both top-down and bottom-up design methodologies. The internal details of an entity are described by an architecture body. Any architecture is associated with only one entity, but a single entity can have multiple architectures.

In general, the modeling style of a model can be:

- structural;
- behavioral;
- mixed.
In structural modeling, an entity is modeled as a set of components connected by signals. In the behavioral modeling, however, the entity is modeled by statements describing the functionality of the device. When the description of an entity contains both structural and behavioral model elements, it is called mixed level modeling.

It should be mentioned that another popular set of modeling styles includes data flow modeling in addition to the ones described above. The definition of behavioral modeling in this case, however, deviates from the definition we gave previously. Here, the behavioral modeling specifies the behavior of an entity as a set of statements that are executed sequentially, and the data flow modeling specifies the functionality of the entity by using concurrent signal assignment statements. These definitions relate the modeling with coding style, i.e., whether a functionality is expressed sequentially or concurrently. We will think both of these as behavioral modeling since they both describe the FUNCTIONALITY of an entity.

Our modeling philosophy is that behavioral modeling and structural modeling can be coexist at each level through the hierarchy of the system except the very bottom level elements, which can only be a pure behavioral model component. The existence of a behavioral model provides for rapid functional simulation, while the existence of a structural model allows for final architectural considerations. We will elaborate these concepts in Chapter 5 through Chapter 9, in which the implementation of VHDL for MIP is discussed.

We will discuss in next section the definitions of signal and variable, and other concepts related with them.

### 3.3 Signal vs. Variable

A signal is an object that has a past history of values, a current value, and a set of future values. A variable, on the other hand, is an object which holds a single value of a given type. Signal objects can be regarded as end points of wires in a circuit. The information a signal object carries is a two-dimensional waveform: The change of its digital states vs. the change of the time. In order to discuss signal and variable objects in greater detail, we need to introduce some related concepts.

**Event:** a term to indicate the change of a signal's value at a specified simulation time.

An event for a signal occurs at the simulation time if the value of the signal changes.
Otherwise, an event does not occur at the simulation time.

**Process:** the basic unit of execution. The unit contains sequential statements that describe the functionality of a portion of an entity. A process statement itself is a concurrent statement. More than one process can be used within an architecture body to capture the behavior of interacting processes.

**Sensitive List:** a set of signals to which the process is sensitive. Any time an event occurs on any signal in the sensitive list, the statements in the process will be executed sequentially. The process suspends after execution of last sequential statement in the process and waits for another event on any signal in the sensitive list to occur.

**Delta Delay:** a representation of an infinitely small delay. This small delay corresponds to a zero time delay of a device and hence does not correspond to any real simulation time. Each unit of simulation time can be considered being composed of an infinite number of delta delays. The purpose of delta delay is to provide a mechanism for ordering events on signals that occur at the same simulation time. Therefore, an event on a signal always occurs at a real simulation time plus an integral number of delta delays.

**Inertial Delay:** the amount of delay time for a stable signal to propagate from input to output of a concurrent element, or statement. If the input signal is not stable during the specified inertial delay time, no event for the signal will be scheduled. Inertial delay is often used to filter out unwanted spikes and transients on signals. Since the inertial delay is mostly common in digital circuits, it is the default delay model.

**Transport Delay:** the amount of delay time for a signal to propagate from input to output of a concurrent element, or statement. It models pure propagation delay. Transport delay model is especially useful for modeling wire delays. Any input pulse, no matter how small its width, will be propagated to output after the specified delay time.

The concepts explained above are important to understand the differences between a signal and a variable. First of all, a variable is different from a signal in terms of the value assignment. A variable is always assigned a value immediately upon evaluations, but a signal is assigned a value after the specified delay or a delta delay. Secondly, a signal has value and time information, but a variable has value only. Thirdly, processes within
an architecture body communicate with each other using signals that are visible to all the processes. However, variables can not be used to pass information between processes since their scope is limited to within a process.

When a signal is assigned a value inside of a process, it uses a sequential signal assignment statement; when a signal is used outside of a process, it uses a concurrent signal assignment statement. A concurrent signal assignment statement is event-driven so it is executed whenever there is an event on a signal that appears in its defining expression. A sequential signal assignment is not event-driven and is executed in the order determined by the sequential list of statements in a process.

Although there are many other important concepts in VHDL, we have only chosen the very small number of concepts to be discussed in previous sections since we feel that understanding of these concepts is vital to our system modeling and simulation. In next chapter, we will describe the functionality of the Morphological Image Processor.
Chapter 4

System Description

4.1 Data Path

In the previous chapter, the important concepts of VHDL have been introduced and discussed. This chapter will describe the functionality of the MIP. As an image processing sub-system, the MIP is able to:

- receive commands from the PC;
- receive and store an image from the PC;
- process an image;
- allow the PC to retrieve a processed image.

The transfer of data and commands is accomplished between the Extended Industrial Standard Architecture (EISA) bus on the PC and local buses on the MIP, while the image processing is done in the ALU1, the MU, and the ALU2. The detailed information is shown in figure 4.1. The dark colored paths in the figure are the local buses of the system. The paths with light color are the connective wires. The data transactions and image processing can be described in four stages:

Load Data: image data are written from the PC to the memory bank through the EISA bus and the I/O bus. Mask data are written from the PC to mask registers in MU through EISA bus.
Figure 4.1: Any image of appropriate size can be written to (or read from) the memory bank from (or to) the PC memory. The morphological mask values are written directly from the PC to registers in the MU. The image can be processed by ALU1, MU, and ALU2. The resultant image from ALU2 is stored back to the memory bank. The volume adder sums each pixel value in a frame and send the result to the PC.
**Process Image:** during the MIP operation, ALU1 uses an image from the memory bank through X1 bus as one operand and an image through X2 bus as the other operand. The output of the ALU1 is entered into the MU, which then processes the image and sends out the resultant image as well as the original image to ALU2. ALU2 selects two of the three input images as its operands. Two inputs are from the MU, and the third input is from the X2 bus. For each processing cycle, the image from the X2 bus can only be exclusively used by either ALU1 or ALU2.

**Store Output:** after the MIP operation, the processed image is stored back to the memory bank through Y bus. The volume of the image is stored in the Volume Adder registers which are read directly by the PC.

**Read Out:** the image data in the memory bank can be retrieved by the PC through the X1, I/O, and the EISA buses.

The blocks shown in figure 4.1 are the functional data blocks for the MIP. The control mechanism is not included in the figure in order to concentrate on the data-flow portion of the MIP.

### 4.2 Arithmetic Functional Blocks

The arithmetic functions of the MIP are accomplished by two ALUs, the MU, and the Volume Adder. The functionality of each component is explained in the following sections.

#### 4.2.1 ALUs

There are two ALUs. ALU1 is the pre-processor for the MU, while ALU2 is the post-processor for the MU. As the name suggests, the ALUs perform arithmetic and logic operations. ALU1 uses data from the X1 and X2 buses as its operands, while ALU2 uses data from two of its three input ports as its operands. These input ports are: the original and processed images from the MU, and the image from any of the four memories through the X2 bus. We will refer to the two active input ports in both ALU1 and ALU2 as A and B in future discussions.

The operation of an ALU depends on the 4-bit binary code shown in table 4.1. The operations for the ALU1 and the ALU2 are very similar. The only difference is that the 4x
and 5x hex coded operations are able to generate the maximum/minimum value of a image on ALU2 but not on ALU1. Each operation is described below:

0x: compares the corresponding pixels of images A and B, and outputs an image composed of the pixels with the minimum value of the comparison.

1x: compares the corresponding pixels of images A and B, and outputs an image composed of the pixels with the maximum value of the comparison.

2x and 6x: copies the input image from A as the output image.

3x and 7x: copies the input image from B as the output image.

4x: searches for the minimum value of a pixel in image A and stores the result in a register; it also copies the input image from A as the output image.

5x: searches for the maximum value of a pixel in image A and stores the result in a register; it also copies the input image from A as the output image.

8x: subtracts image B from image A. If the value of a pixel in B is $-\infty$, then the output value of that pixel is $-\infty$.

9x: subtracts a constant from image A. If the value of the constant is $-\infty$, then the whole output frame value is $-\infty$.

Ax and Ex: adds image A with image B.

Bx and Fx: adds a constant value to all pixels in image A.

Cx: subtracts image B from image A. If the value of a pixel in B is $-\infty$ then the output value of that pixel is the value in A.

Dx: subtracts a constant value from all pixels in image A. If the value of the constant is $-\infty$, then image A is used as the output.
### ALU OPERATIONS

<table>
<thead>
<tr>
<th>Bit</th>
<th>Op</th>
<th>ALUs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hex</td>
<td></td>
<td>ALU1</td>
</tr>
<tr>
<td>0x</td>
<td>0 0 0 0 min(A,B)</td>
<td>YES</td>
</tr>
<tr>
<td>1x</td>
<td>0 0 0 1 max(A,B)</td>
<td>YES</td>
</tr>
<tr>
<td>2x</td>
<td>0 0 1 0 copy(A)</td>
<td>YES</td>
</tr>
<tr>
<td>3x</td>
<td>0 0 1 1 copy(B)</td>
<td>YES</td>
</tr>
<tr>
<td>4x</td>
<td>0 1 0 0 min(A),copy(A)</td>
<td>NO</td>
</tr>
<tr>
<td>5x</td>
<td>0 1 0 1 max(A),copy(A)</td>
<td>NO</td>
</tr>
<tr>
<td>6x</td>
<td>0 1 1 0 copy(A)</td>
<td>YES</td>
</tr>
<tr>
<td>7x</td>
<td>0 1 1 1 copy(B)</td>
<td>YES</td>
</tr>
<tr>
<td>8x</td>
<td>1 0 0 0 A-B</td>
<td>YES</td>
</tr>
<tr>
<td>9x</td>
<td>1 0 0 1 A-const</td>
<td>YES</td>
</tr>
<tr>
<td>Ax</td>
<td>1 0 1 0 A+B</td>
<td>YES</td>
</tr>
<tr>
<td>Bx</td>
<td>1 0 1 1 A+const</td>
<td>YES</td>
</tr>
<tr>
<td>Cx</td>
<td>1 1 0 0 A-B (2)</td>
<td>YES</td>
</tr>
<tr>
<td>Dx</td>
<td>1 1 0 1 A-const (2)</td>
<td>YES</td>
</tr>
<tr>
<td>Ex</td>
<td>1 1 1 0 A+B</td>
<td>YES</td>
</tr>
<tr>
<td>Fx</td>
<td>1 1 1 1 A+const</td>
<td>YES</td>
</tr>
</tbody>
</table>

Table 4.1: ALUs' Operations

#### 4.2.2 MU and Volume Adder

The Morphology Unit performs erosion or dilation as defined by grey-scale morphological operations. It adds the value on the mask with the corresponding pixel in sub-array of the image defined by the target pixel. The minimum value for the target pixel is chosen from the erosion operation or the maximum value is chosen from the dilation operation.

The Volume Adder takes the output from ALU2 and sums either the squared value or the absolute value of each pixel to produces the volume of each image. The output is stored in registers which can be accessed from the PC.

Although not included in figure 4.1, the control mechanism is important for the proper operation of the MIP. The control and status registers are essential to accomplish this task. The functionality of these registers are described in the next section.
4.3 Control and Status Registers

Table 4.2 shows the locations and addresses of the on board control or status registers. The control and status registers will be described in terms of their functionalities. In each of the tables used in this section, the ADDR column shows the hexadecimal register address accessed through the PC bus. The BITS column indicates the corresponding bits for a certain function. The R/W column shows that the register is either read-only by the PC or write-only from the PC. If an address is shown as both readable and writable, the address is actually shared by a read-only and a write-only register. The Content/Purpose column describes the content of the read-only registers and the purpose for the write-only registers.

<table>
<thead>
<tr>
<th>LOCATION</th>
<th>REG No.</th>
<th>ADDR</th>
<th>LOCATION</th>
<th>REG No.</th>
<th>ADDR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Volume Adder</td>
<td>0</td>
<td>000300</td>
<td>ALU2</td>
<td>7</td>
<td>000307</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>000301</td>
<td></td>
<td>8</td>
<td>000308</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>000302</td>
<td>Master Controller</td>
<td>9</td>
<td>00030B</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>000303</td>
<td></td>
<td>10</td>
<td>00030C</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>000304</td>
<td></td>
<td>11</td>
<td>00030D</td>
</tr>
<tr>
<td>ALU1</td>
<td>5</td>
<td>000305</td>
<td>Bus Interface</td>
<td>12</td>
<td>00030E</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>000306</td>
<td></td>
<td>13</td>
<td>00030F</td>
</tr>
</tbody>
</table>

Table 4.2: The Map of Registers in the MIP

4.3.1 Volume Adder Registers

Bit 0 of the Reg.0 selects the original or squared output from ALU2 as the input of Volume Adder as shown in table 4.3. The 34-bit volume is stored in read-only registers Reg.4 through Reg.0 in absolute binary format. Bit 7 of the Reg.4 indicates that a negative value has been passed into the Volume Adder.

4.3.2 ALUs

Table 4.4 shows the control and status registers for the operations of the ALUs. Each ALU requires 9 bits to store a constant in two's complement format. ALU1 uses Reg.5 and bit 0 of Reg.6 to store a constant while ALU2 uses Reg.7 and bit 0 of Reg.8. Bit 7 through bit 4 of Reg.6 and Reg.8 are used to store op-codes for ALU1 and ALU2 respectively. Bit
Table 4.3: Volume Adder Registers

3 and bit 2 of Reg. 8 select the inputs of the A and B operands for ALU2. The minimum or maximum value for operation 4x or 5x in table 4.1 can be read from Reg. 7 and bit 0 of Reg. 8.

Table 4.4: ALU Registers

4.3.3 Master Controller

Bit 0 of Reg. 9 shown in table 4.5 is an active low signal to start the MIP processing.

Table 4.5: Start Register
Table 4.6 shows the control and status register for the MIP. Bit 7 and bit 6 of Reg.10 are used for pipelined operation. When bit 6 is active high, it indicates that the MIP processing is running, and the system is safe to load the new mask or the new image. Bit 7 indicates that the MIP processing is done and the system is safe to start the new processing. Bit 1 of Reg.10 is used to select the X2 bus connection to either ALU1 or ALU2. This selection must match the ALU operations. Bit 0 is used to select either erosion (when it is 0) or dilation (when it is 1).

<table>
<thead>
<tr>
<th>REG</th>
<th>ADDR</th>
<th>BITS</th>
<th>R/W</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>00030C</td>
<td>7</td>
<td>R</td>
<td>OK to start next run</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6</td>
<td>R</td>
<td>OK to load next instruction/window</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>W</td>
<td>bus mode (0=X2-Bus→ ALU1, 1=X2-Bus→ ALU2)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>W</td>
<td>MU’s max (0=min, 1=max)</td>
</tr>
</tbody>
</table>

Table 4.6: Control and Status Registers

Reg.11 in table 4.7 is used to configure the MIP bus connections in either local or the PC mode. If bits 7 to 4 are all zeros, the MIP is in PC mode and one of the memories in the memory bank should be chosen, using bits (3 to 0). This will connect one of the memories with the PC through the X1 bus for a memory read. Otherwise, the MIP is in local mode, and all 8 bits are used to configure the connection between one of the four memories and the local buses X1, X2, and Y for MIP processing.

4.3.4 Bus Interface

Reg.12 in Table 4.9 is used to configure the memory segments. Bit 5 of Reg.12 is a flag for a mask load or a memory access. When the flag is 0, bits 4 and 3 of Reg.12 select the memory controller which uses the address from the PC bus for either a memory read or write. When the flag is 1, no memory controller is selected and each memory uses the address generated by its own memory controller.

The PC address control is detailed in table 4.10. Bit 6 of Reg.13 enables write capability to the on-board memories and mask registers when it is set to 1. Its default value is 0. Bits 4 through 0 of Reg.13 set up the source address for a memory write or the destination.
### MEMORY SELECT: PC MODE

<table>
<thead>
<tr>
<th>REG</th>
<th>ADDR</th>
<th>BITS</th>
<th>R/W</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>00030D</td>
<td>7-4</td>
<td>W</td>
<td>0000 for PC model setup</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>W</td>
<td>connect memory 3 to PC</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>W</td>
<td>connect memory 2 to PC</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>W</td>
<td>connect memory 1 to PC</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>W</td>
<td>connect memory 0 to PC</td>
</tr>
</tbody>
</table>

### MEMORY SELECT: LOCAL MODE

<table>
<thead>
<tr>
<th>REG</th>
<th>ADDR</th>
<th>BITS</th>
<th>R/W</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>00030D</td>
<td>7,6</td>
<td>W</td>
<td>memory 3 to local bus connect (see Table 4.8)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5,4</td>
<td>W</td>
<td>memory 2 to local bus connect (see Table 4.8 )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3,2</td>
<td>W</td>
<td>memory 1 to local bus connect (see Table 4.8 )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1,0</td>
<td>W</td>
<td>memory 0 to local bus connect (see Table 4.8 )</td>
</tr>
</tbody>
</table>

Note: refer to Table Y if bits 7:4 are all zeros

**Table 4.7: Memory Select**

### Table 4.8: Memory to Local Bus Connection

<table>
<thead>
<tr>
<th>BITS</th>
<th>SELECT</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2 \times I + 1$</td>
<td>$2 \times I$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

I is the memory No.

**Table 4.8: Memory to Local Bus Connection**

### Table 4.9: On-board Memory Segments

<table>
<thead>
<tr>
<th>REG</th>
<th>ADDR</th>
<th>BITS</th>
<th>R/W</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>00030E</td>
<td>5</td>
<td>W</td>
<td>mask load select: 0 → memory load, 1 → mask load</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4,3</td>
<td>W</td>
<td>memory controller select = $bit(4) \times 2 + bit(3)$</td>
</tr>
</tbody>
</table>

**Table 4.9: On-board Memory Segments**
address for a memory read. This address information will be compared with the addresses on SA bus to determine whether the address on SA bus is valid.

<table>
<thead>
<tr>
<th>REG</th>
<th>ADDR</th>
<th>BITS</th>
<th>R/W</th>
<th>Purpose</th>
</tr>
</thead>
</table>
| 13  | 00030F | 6    | W   | write enable for on board memory and mask registers  
0 → write disable, Power On Default,  
1 → write enable                                      |
| 4-2 |        | W    | PC BASE ADDR:  
(bit(4) × 4 + bit(3) × 2 + bit(2)) × 200000H                                                     |
| 1,0 |        | W    | PC OFFSET ADDR:  
bit(1) × 100000H + bit(0) × 080000H                                                               |

Table 4.10: PC Address Control

4.4 EISA Bus Interface

The MIP board communicates with a PC computer through the EISA bus. The EISA bus includes a 16-bit data bus, a 24-bit memory address bus, and various control signals. The operation of the MIP requires only a subset of the whole bus protocol. The required signals are: \( \text{RESET DRV, BCLK, BALE, } SA(0 : 19), \ LA(17 : 23), \ SD(0 : 15), \ MEMCS16, \overline{MEMW}, \overline{MEMR}, \overline{IOW}, \) and \( \overline{IOR}. \) The following series of definitions gives detailed functional descriptions of the signals mentioned above.

\( \text{RESET DRV:} \) \( \text{RESET DRV} \) (reset driver) is an output signal that is held active high during system power-on sequences. It remains high until all levels have reached their specified operating range; then it goes inactive low. In addition, the \( \text{RESET DRV} \) line is brought active high if any power level falls outside its specified operating range after a power on. This signal is called \( \text{RESET} \) on the MIP and is used to provide a power-on reset to bring the MIP to a known state before its operation.

\( \text{BCLK:} \) The \( \text{BCLK} \) (bus clock) signal is an output signal providing an 8 MHz clock frequency for the MIP.

\( \text{BALE:} \) The \( \text{BALE} \) (bus address latch enable) is an output signal. This signal goes active high prior to the address bus being valid and falls to inactive low after the address bus is valid. It is used to latch the address information for the MIP.
SA(19:0): Address bits SA19 through SA0 are output signals used to address the system-bus attached memory and I/O. These signal lines are driven during system-bus cycles for memory read, memory write, I/O read, and I/O write operations.

LA(23:17): Unlatched address bits LA23 through LA17 are output signals used to provide memory address information about the present bus cycle. These address signals, unlike SA19-SA0, are only valid for small portion of the addressing cycle. The information provided by these address signals are latched on the falling edge of the BALE signal by the MIP.

SD(15:0): Data bits SD15 through SD0 are bidirectional signals that support the transfer of data between the computer and the MIP.

MEMCS16: The MEMCS16 active low signal is used to indicate the 16-bit data transfer on the present bus cycle. The signal is called MEMCS16n on the MIP.

MEMW: The MEMW signal is an active low output signal used to write data from the system bus into memory. The signal is called MEMWn on the MIP.

MEMR: The MEMR signal is an active low output signal used to request data from the memory. The signal is called MEMRn on the MIP.

IOW: The IOW signal is an active low output signal. It indicates that the address bus contains an I/O port address and the data bus contains data to be written into the I/O register of the MIP. The signal is called IOWn on the MIP.

IOR: The IOR signal is an active low output signal. It indicates to the I/O port that the bus cycle is an I/O port-read cycle and the address bus contains an I/O port address. The I/O register on the MIP will respond by placing its data on the system data bus. The signal is called IORN on the MIP.

The timing information for the above signals can be summarized by reset, I/O read, I/O write, memory read, and memory write operations. Figure 4.4 shows the signal timing for the reset bus cycle. The RESET pulse lasts for 1250 ns during which all the other signals are disabled. 60 ns after RESET becomes inactive, BCLK starts to generate clock pulses. The clock frequency is about 8 Mhz. Figure 4.3 shows the signal timing of the register write/read bus cycles. The starting 0 ns point indicates the time at which the command
starts. The only difference between the read and write cycles is the timing on the data bus. For a write operation, data is valid after \( IOW \) becomes active low and stays valid till \( IOW \) becomes inactive. For a read operation, the data is required to be valid some time before and after \( IOR \) becomes inactive. This is determined by the setup time and the hold time. Figure 4.4 shows the signal timing of the memory write/read bus cycles. The timing diagrams are based on simulation results from the PCBUS.BLM model written by Jeff Hanzlik [8]. The above operations are controlled by the system through a series of I/O commands. These commands are listed in table 4.11.

Figure 4.2: Signal Timing of Reset Bus Cycle

![Diagram](image)

In next section, we will discuss how to use these commands for an operation.

4.5 Operating Procedures

The system commands described in the previous section are used to control the MIP’s operation. It is noticed in the simulation of the MIP that the execution sequence of commands
Figure 4.3(a): Signal Timing of Register Write Bus Cycle

Figure 4.3(b): Signal Timing of Register Read Bus Cycle
Figure 4.4(a): Signal Timing of Memory/ Mask Write Bus Cycle

Figure 4.4(b): Signal Timing of Memory Read Bus Cycle
<table>
<thead>
<tr>
<th>Commands</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reset &lt;address&gt;</td>
<td>causes RESET signal on the PC bus being active</td>
</tr>
<tr>
<td>IOR &lt;address&gt;</td>
<td>reads 8-bit data from MIP’s output register to specified I/O port</td>
</tr>
<tr>
<td>IOW &lt;address&gt;</td>
<td>writes 8-bit data from specified I/O port to MIP’s input register</td>
</tr>
<tr>
<td>PICLD &lt;address&gt; &lt;image file&gt;</td>
<td>writes 9-bit data of a frame from the specified image file to the memory</td>
</tr>
<tr>
<td>MASKLD &lt;address&gt;</td>
<td>writes 9-bit data of a window to the mask.</td>
</tr>
<tr>
<td>PICRD &lt;address&gt;</td>
<td>reads 9-bit data of an image from MIP to the system</td>
</tr>
</tbody>
</table>

Table 4.11: System Commands

is very important. The sequence is shown in figure 4.5 in the form of flow chart which illustrates the MIP's operation. The grey blocks in the figure represent the stages that the MIP's operation will step through. Each of the dotted blocks is an operating unit which contains a sequence of necessary register and memory operations represented by solid black rectangles. The following subsections are to discuss each of the stages in the MIP operation.

4.5.1 Stage A: Memory / Mask Write

The procedure starts by writing the image data and mask values into the MIP board. In theory, the Memory write and Mask write can be executed in arbitrary order. In practice, however, it is necessary to first configure the MIP to load the mask before starting the MIP process due to the design constrains.

Memory write: writes an image frame to an on-board memory. Reg.15 and Reg.14 must be set up before the operation.

- **IOW 00030F <data field>** enables the memory and mask write, and sets the PC memory source address (base and offset) for the image.
- **IOW 00030E <data field>** configures memory/mask flag to memory load, and selects the memory controller which controls the memory receiving the data.
- **PICLD <address>** writes the image frame to the selected on-board memory starting at <address+2>.
Figure 4.5: MIP Operation Flow Chart

A: Memory/ Mask Write -
Write to on board memory or mask according to addresses provided by PC bus.

B: ALUs and MAP Setup -
Setup constants and operations in ALU1, MAP, and ALU2.

C: MIP Process -
Configure memory to local buses; start ALU1, MAP, and ALU2 operations.

D: Memory/RegistersRead -
Read the on board memory or registers’ values to PC memory according to addresses provided by PC bus.
**Mask write:** writes a mask to MU. Reg_15 and Reg_14 must be configured before the operation.

- **IOW 00030F** <data field> enables memory and mask write, and sets PC memory source address (base and offset) for the mask.
- **IOW 00030E** <data field> sets memory/mask flag to mask.
- **MASKLD** writes mask to registers in MU.

### 4.5.2 Stage B: ALUs and MU setup

After loading the memory and the mask, the ALU operations, the MU operation, and the local bus configurations can be set up in arbitrary order.

**ALU1 SETUP:** sets up ALU1 constant and operation by the following commands:

- **IOW 000305** <data field>
- **IOW 000306** <data field>

**ALU2 SETUP:** sets up ALU2 constant and operation; connects the ALU2 A input to either X2 bus or MU’s Y output; connects the ALU2 B input to either X2 bus or MU’s X output. The commands are:

- **IOW 000307** <data field>
- **IOW 000308** <data field>

**MU and X2 BUS SET UP:** sets up the X2 bus connection to ALU1 or ALU2 and the maximum or minimum operation of MU by the command:

- **IOW 00030C** <data field>

### 4.5.3 Stage C: MIP Process

At this stage, the local bus mode must be set up before starting the MIP process.

- **IOW 00030D** <data field> configures the connections between memory and local buses.
- **IOW 00030B** starts the MIP operation.
• **SKIP**: skips the number of pc bus clock cycles determined by the MIP board before the resultant image can be fetched by the system.

### 4.5.4 Stage D: Memory/Register Read

After the MIP processing is done, the resultant image and the volume can be read back to the PC in arbitrary order.

**Memory Read**: reads the image from MIP to PC.

- **IOW 30E <data field>** configures memory/mask flag to memory load, and selects the memory controller which controls the memory sending the data.
- **IOW 30D <data field>** configures the connection between the on-board memory and the PC memory.
- **PICLD** reads the memory.

**Register Read**: reads registers 000300-304,000307,000308 and 00030C.

- **IOR <address>**
Chapter 5

Architecture Partition and Modeling

The data path of the MIP has been described in Chapter 4. In this chapter we will discuss the architecture of the MIP as well as the behavioral and the structural models of the entire MIP system.

5.1 Architecture Partition

Figure 5.1 shows the architectural hierarchy of the MIP. The hierarchy is based on the functional blocks of the MIP. A functional block identified with a square indicates an actual circuit component encapsulated by a VHDL model. A functional block identified with an ellipse is a virtual component used for classification. We will briefly describe the functionality of each block accordingly.

The Morphological Image Processor is a system model for the whole MIP. Two independent models are designed to emulate the system for different usages: a stand-alone behavioral model is used to emulate the system’s behavior using minimum simulation resources; a structural model is constructed using lower level functional blocks carrying the architecture information. The structural model consists of four functional blocks: I/O Unit, Control Units, Arithmetic Units, and Memory Units.

The I/O Unit is the interface between the host computer and the MIP board. It accepts commands from the host computer and distributes the commands to the other units.

The Control Units enclose both the Master Controller and the Memory Controller.
Figure 5.1: Architectural Hierarchy of Morphological Image Processor

- **Morphological Image Processor**

- **I/O Unit**
  - PC Bus Interface (Hao)
  - Master Controller (Hao)
  - Memory Controller (Hao)

- **Controll Units**

- **Arithmetic Units**
  - ALU1 (Wei-chun)
  - ALU2 (Wei-chun)
  - MU (Wei-chun)
  - Volume Adder (Wei-chun)

- **Memory Units**
  - Memory (Wei-chun)
  - Buffer (Wei-chun)

**Legend**
- (): Author
- (): Functional Block
- (): Physical Block
The Master Controller is responsible for timing and blanking of the MAP's operation, while a Memory Controller generates the correct address for the corresponding memory during either read or write memory cycle.

The Arithmetic Units includes four functional blocks: ALU1, MU, ALU2, and Volume Adder. These units perform the computing functions described in the previous chapter. The MU consists of the MAP and the FIFO. FIFOs are used to provide the line delay for the input to the MAP. The MAP performs the morphological operations described in the previous chapter.

The Memory Units include the Memory which is an on-board memory chip, and Buffer which is a tri-state I/O buffer. The on-board memory is used to store the original image or a processed image, while a buffer is used to control the signal flow.

In order to implement the functionality of the MIP in hardware, Jeff Hanzlik and Jens Rodenberg designed the original architecture and implemented a prototype board using Field Programmable Gate Arrays (FPGA). The FPGA is suitable for prototype circuit design because of its low cost and fast turn out. After the architecture was verified by the FPGA version, Larry Rubin, Chris Insalaco, and Shishir Ghate began a design to implement the same architecture with fully customized VLSI devices to improve the circuit speed of the process.

The differences between the FPGA version and the VLSI version are:

1. the memory controller chip contains one memory controller in FPGA version, but two in the VLSI version.
2. the MAP is composed of 26 chips in FPGA version, but 7 chips in the VLSI version.
3. ALU1 and ALU2 are identical devices in the VLSI version.
4. the image size in the FPGA version is fixed at 512 x 512. However, the image size in the VLSI version can be either 1024 x 1024 or 512 x 512.

In the VHDL models, the architecture has been partitioned according to the functionality of the MIP shown in Figure 5.1. These functional blocks will be suitable for applying different technologies with the same architecture through synthesis or manual conversion without modifying the architecture.
<table>
<thead>
<tr>
<th>Ports</th>
<th>I/O</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RESET</td>
<td>I</td>
<td>SYSTEM RESET, active high</td>
</tr>
<tr>
<td>BCLK</td>
<td>I</td>
<td>EISA bus clock</td>
</tr>
<tr>
<td>BALE</td>
<td>I</td>
<td>Bus address available, active high</td>
</tr>
<tr>
<td>LA(23:17)</td>
<td>I</td>
<td>Base address bus</td>
</tr>
<tr>
<td>SA(19:0)</td>
<td>I</td>
<td>Segment address bus</td>
</tr>
<tr>
<td>SD.in(15:0)</td>
<td>I</td>
<td>Data input bus*</td>
</tr>
<tr>
<td>IORn</td>
<td>I</td>
<td>Registers read, active low</td>
</tr>
<tr>
<td>IOWn</td>
<td>I</td>
<td>Registers write, active low</td>
</tr>
<tr>
<td>MEMRn</td>
<td>I</td>
<td>Memory read, active low</td>
</tr>
<tr>
<td>MEMWn</td>
<td>I</td>
<td>Memory/Mask write, active low</td>
</tr>
<tr>
<td>SD.out(15:0)</td>
<td>O</td>
<td>Data output bus*</td>
</tr>
<tr>
<td>MEMCS16n</td>
<td>O</td>
<td>8/16 data transmit mode, active low</td>
</tr>
<tr>
<td>Frame(1:0)</td>
<td>I</td>
<td>Applied frame size mode</td>
</tr>
</tbody>
</table>

Table 5.1: MIP Entity

5.2 **Behavioral Model of the MIP**

Both behavioral and structural models of the MIP emulate the MIP system described in the previous chapter. A testbench should be provided in VHDL to test a model. Unfortunately, the testbench requires the file I/O capability which is not implemented in our design tool, Mentor Graphics' system 1076 version 7.0. In order to read the system's commands, input images, and store the output results, we used the BLM model of PC BUS written by Jeff Hanzlik as the testbench. Figure 5.2 shows the schematic in which the BLM model of PC BUS and the VHDL model of the MIP are connected by Mentor Graphics' Neted. The I/O ports in the entity are shown in table 5.1. It should be realized that the SD.in and SD.out ports are actually a bi-directional bus SD on EISA bus. Since the system 1076 version 7.0 does not support the INOUT port type, we decided to split the SD bus into SD.in and SD.out, and connected these two ports outside the VHDL models. The port Frame is used to configure the image size which can be smaller than $512 \times 512$ for the simulation purpose. The actual hardware does not include this port.

The design goal of MIP's behavioral model is to obtain simulation results to compare with the design specifications of the MIP. The processes in the model are designed based on:

1. the sources and destinations of the system commands in table 4.11. The processes
Figure 5.2: MIP Behavioral Model with PC BUS
are: mip._IOW_.process for register write, mip._MEMW_.process for memory and mask write, and mip._SD_output_.process for register and memory read.

2. the morphological operation. The processes are mip._MAP_.process for all of the image processing operations and mip._pipeline_.process for the status signals of pipelined-processing.

3. the bi-directional signal emulation. The processes are mip._SD_in_ and mip._SD_out_.

4. the bus I/O conversion. "Convert" at the end of the process’ name is used to indicate the conversion feature of the process.

5.2.1 SD Bus Emulation

The SD is a bi-directional tri-state data bus in the EISA bus protocol. A port connects with SD through a pair of tri-state buffers (the behavioral model of the tri-state buffer and the resolving function are discussed in chapter 7). In a fully implemented 1076 VHDL system, this port should be defined as a resolved tri-state signal with a INOUT port type. In this version, the ports SD_in and SD_out are used to mimic the SD bus for input and output respectively. As shown in the schematic, these two ports are connected outside the VHDL model and resolved by PCBUS.BLM. The SD_BUF_IN is the uni-direction buffer for SD_in. It is turned on when either the IOWn or the MEMWn is active. On the other hand, the SD_BUF_OUT is the uni-direction buffer for SD_out. It is turned on when either the IORn, or the MEMRn is active.

```
518     mip_sd_in:
519     PROCESS
520     BEGIN
521         wait on SD_in, IOWn,MEMWn, RESET;
522         if IOWn='O' or MEMWn='0' then
523             SD_BUF_IN <= SD_in;
524         else SD_BUF_IN <= "ZZZZZZZZZZZZZZ";
525         end if;
526     END PROCESS mip_sd_in;

527     mip_sd_out:
528     PROCESS
529     BEGIN
530         wait on SD_BUF_OUT, IORn, MEMRn, RESET;
531         if IORn='O' then
```
5.2.2 Bus to Integer Conversion

Ideally the on-board memory can be modeled by using the qsim_state_vector for the address and content signals. The qsim_state is a non-standard type defined by Mentor Graphics. There are four logic states in qsim_state: '0' and '1' for logic 0 and 1, 'X' for unknown state, and 'Z' for hi-impedance state. The qsim_state_vector is a type which is an unconstrained array of qsim_state. The data structure of the memory requires an array of qsim_state_vector. Unfortunately the two-dimensional array type is not supported by system 1076 version 7.0. Therefore, the data structure of the on-board memory and mask registers must use an one dimensional integer array. The address signals, LA and SA, are converted into integers as the index of the memory array while the data signals, SD_in and SD_out, are converted into integers as the memory contents.

The following process, mip_SD_read_and_convert is one of the converting processes. The process converts the vector, SD_BUF_IN, into an integer signal, SD_IN_REG, whenever an event occurs on the SD_BUF_IN.

The in_gen procedure converts a qsim_state_vector into an integer.

```
mip_SD_read_and_convert:
PROCESS
VARIABLE temp_value: integer;
VARIABLE temp_signal: qsim_state_vector(8 downto 0);
BEGIN
wait on SD_BUF_IN;
for i in temp_signal'LENGTH-1 downto 0 LOOP
  temp_signal(i) := SD_BUF_IN(i);
end loop;
in_gen(temp_signal,temp_value);
SD_IN_REG <= temp_value;
END PROCESS mip_SD_read_and_convert;
```

The in_gen procedure converts a qsim_state_vector into an integer.
For the 9-bit signed integers in the model, the procedure `out_gen` converts them into 16-bit `qsim_state_vectors`.

```plaintext
PROCEDURE out_gen (value : IN integer;
output : OUT qsim_state_vector(15 DOWNTO 0)) IS
  VARIABLE i,choice,temp: integer;
BEGIN
  temp := value;
  IF (temp /= UNKNOWN) THEN
    IF (temp < 0) THEN
      temp := temp + 2**(WORD_LENGTH);
      EXIT;
    END IF;
    FOR i IN 0 TO 15 LOOP
      choice := temp mod 2;
      CASE (choice) IS
        WHEN 1 =>
          output(i) := '1';
          EXIT;
        WHEN 0 =>
          output(i) := '0';
          EXIT;
        WHEN OTHERS =>
          NULL;
        END CASE;
      END LOOP;
    temp := temp / 2;
  ELSE
    FOR i IN 0 TO 15 LOOP
      output(i) := 'X';
    END LOOP;
  END IF;
END out_gen;
```
5.2.3 8/16 Bit Data Transfer

The data transfer between the host computer and the MIP board can be either 8 bits or 16 bits. The signal MECS16n is used to indicate the 16-bit data transfer when it is low. The process `mip_MEMCS16n_process` is used to handle the situation.

```verbatim
mip_MEMCS16n_process:
PROCESS
BEGIN
wait on BALE, BCLK until BALE='0' and BCLK='0';
MEMCS16n <= transport '1' after MEMCS_DELAY;
wait on MEMRn, MEMWn;
if MEMWn='0' or MEMRn='0' then
    MEMCS16n <= transport '0' after MEMCS_DELAY;
end if;
END PROCESS mip_MEMCS16n_process;
```

5.2.4 LA Latching

Each I/O cycle on the PC bus starts when the LA signal becomes valid, which is indicated by the level of BALE as shown in figure 4.3. The following process, `mip_adr_process`, latches the LA address whenever the BALE is '1'.

```verbatim
mip_adr_process:
PROCESS
BEGIN
wait on BALE until BALE='1';
ADDRESS_REG <= LA_REG*2**19;
-- mapping address (23:19)
END PROCESS mip_adr_process;
```

5.2.5 I/O Transfer

After the LA address is valid, any of the four I/O operations (defined in chapter 4.4) can be initiated by the IORn, IOWn, MEMRn, or MEMWn signals. In addition, the RESET signal is used to reset the MIP. The detailed timing information can be found in figure 4.4 and 4.3.
Register Write

The following process is used to either transfer a new value from the PC bus to an on-board input register or to reset the registers. Multiple wait-on statements are included in the process. The process is first invoked by \texttt{IOWn}'s event to generate the address. It is then suspended until the new SD value is available by the converting process shown in section 5.2.2. The case statement is used to update the content of the address specified by \texttt{LA} and \texttt{SA}. The \texttt{MAP\_START}, triggered by address select, invokes the \texttt{mip\_MAP\_process}.

626 mip\_IOW\_process:
627 PROCESS
628    variable data\_temp : integer;
629    variable address\_temp : integer;
630    BEGIN
631       wait on RESET, IOWn until RESET='1' or IOWn='0';
632       if RESET='1' then
633          REG\_30F\_IN <= 0;       -- disable memory write, pc adr=0h
634          REG\_30E\_IN <= 0;       -- mask load, select memory 0
635          REG\_30D\_IN <= 0;       -- don't select any memory for any bus
636          REG\_30C\_IN <= REG\_30C\_IN mod 2;   -- keep the MAP op only (bit 0)
637          REG\_30B\_IN <= 1;       -- reset REGS\_STARTn
638       elsif IOWn='0' then
639          assert FALSE
640          report "write to registers"
641          severity NOTE;
642          address\_temp := ADDRESS\_REG + SA\_REG;
643          wait on SD\_IN\_REG;
644          data\_temp := SD\_IN\_REG;
645          case (address\_temp) is
646             when ADR\_300\_IN =>
647                REG\_300\_IN <= data\_temp;
648             ...  
649             when ADR\_30B\_IN =>
650                MAP\_START <= not MAP\_START;
651                REG\_30B\_IN <= data\_temp;
652             when others =>
653                assert FALSE
654                report "non-exist input register address"
655                severity WARNING;
656            end case;
657       end if;
658    END PROCESS mip\_IOW\_process;
Memory or Mask Write

The address generation and data latch of this process is similar to 5.2.5. However, the process is sensitive to $MEMWn$ instead of $IOWn$. The data latched in this process will be transferred to the mip_MAP_process described in 5.2.6. Before starting memory/mask write, the base address for an image or a mask stored in the PC memory must be loaded into the register, REG_30F_IN. The latched base address in REG_30F_IN is compared with the specified base address in $LA_REG$. The data will not be transferred if the latched address is different from the specified address.

```vhdl
675  mip_MEMW_process: 
676   PROCESS 
677   variable address_temp: integer; 
678   BEGIN 
679   wait on MEMWn; 
680   if MEMWn='0' and MEMWn'EVENT then 
681     address_temp := ADDRESS_REG+ SA_REG; 
682     assert FALSE 
683     report "memory/mask load" 
684     severity NOTE; 
685   wait on SD_IN_REG; 
686   MEMW_BUFFER <= SD_IN_REG; 
687   if (extract_bits(REG_30E_IN,5,5)=MASK_LOAD) then 
688     RAM_ADDRESS <= extract_bits(address_temp,18,1); 
689     assert FALSE 
690     report "memory loading" 
691     severity NOTE; 
692   else 
693     assert FALSE 
694     report "mask loading" 
695     severity NOTE; 
696   end if; 
697   if (LA_REG = extract_bits(REG_30F_IN,4,0)) then 
698     MEMORY_MASK_LOAD <= not MEMORY_MASK_LOAD; 
699   else 
700     assert FALSE 
701     report "non-existing memory address" 
702     severity WARNING; 
703   end if; 
704 end if; 
705 END PROCESS mip_MEMW_process; 
```
Register or Memory Read

In the system 1076 version 7.0, a multi-driven signal is not detected by the system during either the compiling time or the run time. When the situation occurs, the value on the multi-driven signal will be overwritten by the newest signal value. Therefore, special attention is required to resolve a multi-driven signal. In this case, the memory-read and register-read are both processed by `mip_SD.output_process` to avoid a multi-driven `SD.out`. The process is invoked when either `IORn` or `MEMRn` is '0'. If the `IORn` is '0', the register value in the address defined by `address_temp` is passed to `SD.out`. If the `MEMRn` is '0', the `MEMORY_READ` signal will trigger the `mip.MAP.process` to pass the value from memory to `SD.out`.

```
mip_SD_output_process:
PROCESS
  variable address_temp : integer;
BEGIN
  wait on IORn, MEMRn until IORn='0' or MEMRn='0';
  address_temp := ADDRESS_REG + SA_REG;
  if IORn='0' then
    assert FALSE
    report "read from registers"
    severity NOTE;
    case (address_temp) is
      when ADR_300_OUT =>
        SD_OUT_REG <= REG_300_OUT;
      ...
      when others =>
        assert FALSE
        report "non-exist output register address"
        severity WARNING;
        -- should show a warning for illegal register address
  end case;
  elsif MEMRn='0' then
    address_temp := ADDRESS_REG + SA_REG;
    assert FALSE
    report "memory read"
    severity NOTE;
    RAM_ADDRESS <= extract_bits(address_temp,18,1);
    if (LA_REG = extract_bits(REG_30F_IN,4,0)) then
      MEMORY_READ <= not MEMORY_READ;
    wait on BUFFER_READY;
    SD_OUT_REG <= MEMR_BUFFER;
  else
    assert FALSE
```

47
5.2.6 Mathematical Operations

The mip.MAP.process performs the functions of ALU1, ALU2, and MU, as well as the memory transfer. The functions of ALU1, ALU2 and MU have been described in chapter 4. The process performs the pipelined operations differently from the real circuit. In the hardware design the input image is computed stage by stage, starting from ALU1 and ending at the Volume Adder. The sequence can be found in figure 4.1. The intermediate results between stages are stored in temporary buffers. These temporary buffers provide easy access to examine the partial result between the stages for the debugging purpose.

The process can be invoked by four signals: MAP.START from mip.IOW.process when the register 00030B is selected, MEMORY.MASK LOAD from mip.MEMW.process when the system command MEMW is issued, MEMORY.READ from sd.output.process when the command MEMR is issued, and RESET from the PC bus when the whole system is reset. In the following subsections, we will describe mainly the system configuration. The operations of each stage will be presented, but the implementation of the VHDL model for the corresponding physical blocks will be discussed in chapter 6.

Configuration and Operations of the MIP

The operating procedures have been explained in the section 4.5. The process mip.MAP.process checks the memory write enable status, examines the setup of ALUs and MAP, and establishes the local bus configurations. The error messages will be given if the configuration is incorrect.

As shown in stage A of figure 4.5, the memory write status must be confirmed before writing to an on-board memory or the mask.

if (extract_bits(REG_30F_IN,6,6)=1) then -- memory&mask write enable
...
else
  assert FALSE
  report "memory/mask write disable: check 30F"
The implementation of stage B in figure 4.5 is accomplished by retrieving the constants for ALUs and decoding the op-codes for the ALUs, the MAP, and the Volume Adder.

```vhdl
alu1_const := extract_bits(REG_306_IN,0,0)*2**8 + REG_305_IN;
if (alu1_const > MAXNUM) then
  alu1_const := alu1_const - 2**(WORD_LENGTH);
end if;
alu1_op := extract_bits(REG_306_IN,7,4);
...map_op := extract_bits(REG_30C_IN,0,0);
...alu2_const := extract_bits(REG_308_IN,0,0)*2**8 + REG_307_IN;
if (alu2_const > MAXNUM) then
  alu2_const := alu2_const - 2**(WORD_LENGTH);
end if;
alu2_op := extract_bits(REG_308_IN,7,4);
alu2_select := extract_bits(REG_308_IN,3,2);
...volume_op := extract_bits(REG_300_IN,0,0);
```

In the stage C of figure 4.5, memories are connected with the local buses, X1, X2, and Y. The connections are made by copying the contents of the memory into buffers. The better way to emulate the connection would be accessing the memory by a pointer. This can be accomplished by the access type in a fully implemented VHDL. If a memory is connected with X1 bus as the input source, the contents of the memory will be copied to the X1 buffer (x1_buffer).

```vhdl
x1bus_cnt:=0; x2bus_cnt:=0;
bus sel3:=extract_bits(REG_30D_IN,7,6);
case (bus sel3) is
  when 0 => x1_buffer := memory3; x1bus_cnt:= x1bus_cnt+1;
  when 1 => x2_buffer := memory3; x2bus_cnt:= x2bus_cnt+1;
  when others => null;
end case;
...assert not (x1bus_cnt<1)
  report "no memory connects to x1_bus: check 30D"
  severity ERROR;
assert not (x1bus_cnt>1)
  report "more then one memory connect to x1_bus: check 30D"
  severity ERROR;
```
assert not (x2bus_cnt<1)
report "no memory connects to x2_bus: check 30D"
severity WARNING;

assert not (x2bus_cnt>1)
report "more then one memory connected to x2_bus: check 30D"
severity ERROR;

ybus_cnt:=0;
if (bus_sel3=2) then
  memory3 := y_buffer;
ybus_cnt:= ybus_cnt+1;
end if;

assert not(ybus_cnt<1)
report "no memory connects to y_bus: check 30D"
severity ERROR;

assert not(ybus_cnt>1)
report "more then one memory connects to y_bus: check 30D"
severity ERROR;

A MIP user should be cautious not to connect X2 bus to both ALU1 and ALU2. Although this erroneous setup is not prevented by the hardware, our VHDL model provides the check with the following statements.

-- status x2 bus status check
case (alu1_op) is
  when COPY1B|COPY2B|ADDAC1|ADDAC2|SUBAC1|SUBAC2=>
    alu1_x2:=FALSE;
  when others => alu1_x2:=TRUE;
end case;

case (alu2_select) is
  when ALU2_CTOA => -- in_a = x2_bus, in_b = map_xout
    case (alu2_op) is
      when COPY1B|COPY2B => alu2_x2:=FALSE;
      when others => alu2_x2:=TRUE;
    end case;
  when ALU2_CTOB => -- in_a = map_yout, in_b = x2_bus
    case (alu2_op) is
      when MIN1AB|MAX1AB|COPY1B|COPY2B|ADDAB1|
        ADDAB2|SUBAB1|SUBAB2 =>
        alu2_x2:=TRUE;
      when others => alu2_x2:=FALSE;
    end case;
  when ALU2_CAB => -- in_a = x2_bus, in_b = x2_bus
    alu2_x2:=TRUE;
when others => NULL; -- pseudo option
end case;
assert not( alu1_x2 and alu2_x2)
report "bus conflict between alu1, alu2, x2 bus"
severity ERROR;

The stage C in figure 4.5 shows that the image is processed by the ALU1, the MAP, the
ALU2, and the Volume Adder. These operations are executed by the following statements.

alu1_process(x1_buffer,x2_buffer,alu1_const,alu1_op,alu_buffer);
...
map_process(alu_buffer,mask,map_op,APPLIED_ROW_SIZE,map_buffer);
...
case (alu2_select) is
when ALU2_NORM => -- in_a = map_yout, in_b = map_xout
  alu2_process(map_buffer, alu_buffer, alu2_const,
  alu2_op,alu2_max_min, y_buffer);
...
end case;
REG_307_OUT <= extract_bits(alu2_max_min,7,0);
REG_308_OUT <= extract_bits(alu2_max_min,8,8);
...
volume_adder_process(y_buffer,volume_op,volume_reg0,
volume_reg1);
REG_300_OUT <= extract_bits(volume_reg0,7,0);
...
REG_304_OUT <= extract_bits(volume_reg1,15,8);

Memory Write

After the status of memory write enable is checked, either the mask or one of the on board
memories can be written, provided that the configuration is correct.

if (extract_bits(REG_30E_IN,5,5)=MASK_LOAD) then
  -- memory or mask?
  case (extract_bits(REG_30E_IN,4,3)) is -- memory select
    when 3=> memory3(RAM_ADDRESS) := memory_temp;
    ...
    when others => -- should show a warning;
  assert FALSE
  report "illegal memory chip select: check 30E"
  severity ERROR;
end case;
else
  for i in 0 to mask'LENGTH-2 loop -- MASK SHIFT LOAD
    mask(i) := mask(i+1);
  end loop;

end loop;
mask(mask'LENGTH-1) := memory_temp;
end if;

Memory Read

In stage D shown in figure 4.5, the X1 bus is connected with the memory from which the image is read by the host computer. The selected memory in register 00030E must match the selection of X1 bus connection in register 00030D.

if (extract_bits(REG_30D_IN,3,0)/2=extract_bits(REG_30E_IN,4,3)) then
  case (extract_bits(REG_30D_IN,3,0)) is
    when 8=> memory_temp := memory3(RAM_ADDRESS);
    ...
    when others => NULL;
  end case;
else
  assert FALSE
  report "unmatched controller & memory: check 30D and 30E"
  severity WARNING;
  memory_temp := UNKNOWN;
end if;

5.3 MIP Structural Model

The behavioral model discussed above clearly illustrates the functionality of the MIP. However, it does not carry architectural information. In order to see how the functionality of the MIP is implemented in hardware, we must use the structural model, which is shown in figure 5.3. Each block except PCBUS in the figure is the behavioral model of the corresponding circuit. PCBUS is a BLM model written by Jeff Hanzlik to emulate the host computer. Since system 1076 version 7.0 does not support file input and output feature, a VHDL testbench can not be written in a meaningful way. Therefore, we adopted the BLM model of PCBUS into our structural model for simulation purpose. The VHDL models of other components will be discussed in the following four chapters. However, we will discuss the timing and overview of the MIP in this section.
5.3.1 Timing of the MIP

Figure 5.4 shows the general timing of the MIP (FPGA version) provided by Jens Rodenberg. ([9]) The address and data for a pixel are represented by the location of the pixel in row, column format. Timing is shown for one complete image pass, with the relevant pixel locations and control pulses shown in boldface. The last part of a previous image and the beginning of an images following the complete image is shown. All timing waveforms found on the timing diagram are described below. Most of the description are directly from Jens Rodenberg [9]:

\textit{START-MEM(X1)}: originates from the Master Controller. There could be one or more \textit{START-MEM(X1)} signals with identical waveform. The number of signals depends on the bus mode. When the bus mode is 1, only one \textit{START-MEM} signal is generated, i.e, only one memory is selected. The output of the selected memory is connected to the A input of ALU1 through the X1 bus (which is controlled by the \textit{X1-BUS-SEln}). When the bus mode is 0, two \textit{START-MEM} signals are generated. The outputs of selected memories are connected to both A and B inputs of ALU1 through X1 and X2 buses respectively. On the next rising clock edge, the memory address counter in each selected \textit{MEM-CONTROL} chip will start counting. The newly generated addresses (X1 Addr) will be used by the respective memories to output data (X1 data) onto the appropriate data buses during the same clock that addresses are generated. X1 Addr and X1 data denote the addresses and data associated with the memories connected with ALU1.

\textit{X1 Addr}: is the address generated by the \textit{MEM-CONTROL} chip controlling the memory designated to drive the \textit{X1.Bus} when the bus mode is 1. When the bus mode is 0, \textit{X1 Addr} also includes the address generated by the \textit{MEM-CONTROL} chip controlling the memory designated to drive the \textit{X2.Bus}.

\textit{X1 Data}: is the data contained in the selected memories associated with the ALU1 at X1 Addr. The data is clocked into the input flip-flops of ALU1 one clock cycle after the address is generated.

\textit{INIT ALU1}: instructs ALU1 to load its next instruction upon the next rising clock edge. This initializes ALU1 at the same time that the first valid image pixel (0,0) is its
Figure 5.4: MIP Timing Chart
Figure 5.5: MIP Timing Chart (continued)
operand.

**ALU1 Operand**: is the output of the input flip-flops of ALU1. This is the pixel being operated on by ALU1 during any given clock period. The pixel will be clocked into ALU1’s output flip-flops on the next rising clock edge.

**ALU1 Output**: is the output of the output flip-flops of ALU1. The pixel is clocked into the input of the MAP on the next rising clock edge.

**First MAP Operand**: The operand of the first adder in the MAP.

**Start Blank Counter**: instructs the blank counter to start counting on the next rising clock edge. In the VLSI version, the Start Blank Counter occurs one clock period later than in FPGA version. The reason is explained in section B.1.3.

**START PROC**: informs the MAP that the first valid image pixel will be in the target pixel position upon the next rising clock edge. The MAP uses this signal to latch the window values and the desired morphological operation for next image processing operation.

**Blank counter count**: is used to generate the row and column blanking signals.

**Target pixel**: is the pixel being operated on by the adder in the middle of the MAP, which also corresponds to the middle of the window.

**C1 operand**: The pixels in the first level of the comparison tree. There are 49 pixels that are inputs to the first comparison tree level since all 49 adders in the MAP perform a simultaneous addition of a potential result.

**C13 operand**: The pixels in the last level of the comparison tree. The result of the comparison, which is the output value of the MAP, gets clocked into the output flip-flops of the MAP.

**MAP output**: is the output of the output flip-flops of the MAP.

**START_MEM(X1)**: originates from the Master Controller. When bus mode is 1, one **START_MEM** is generated, and the output of the selected memory is connected to the **C** input of ALU2 through the X2 bus (which is controlled by the **X2_BUS_SELn**). On
the next rising clock edge, the memory address counter in selected MEM.CONTROL chip will start counting. The newly generated address (X2 Addr) will be used by the memory to output data (X2 data) onto the X2 data bus during the same clock that address is generated. X2 Addr and X2 data denote the address and data associated with the memory which is connected with ALU2. When the bus mode is 0, no \textit{START\_MEM(X2)} signal is generated.

\textit{X2 Addr}: is the address generated by the MEM.CONTROL chip controlling the memory designated to drive the X2 Bus when the bus mode is 1. When the bus mode is 0, this address will not be generated.

\textit{X2 Data}: is the data contained in the selected memory associated with the ALU2 at X2 Addr when the bus mode is 1. When the bus mode is 0, this data will not be generated. The data is clocked into the input flip-flops of ALU2 one clock cycle after the address is generated.

\textit{INIT ALU2}: instructs ALU2 to load its next instruction upon the next rising clock edge. This initializes ALU2 at the same time that the first valid image pixel (0,0) is its operand.

\textit{STOP ALU2}: informs ALU2 that the last valid pixel of the image being processed is its output on the next rising clock edge. This is used to capture the maximum or the minimum value of the pixels in processed image, if the ALU operation was selected.

\textit{ALU2 Operand}: is the output of the input flip-flops of ALU1. This is the pixel being operated on by ALU2 during any given clock period. The pixel will be clocked into ALU2's output flip-flops on the next rising clock edge.

\textit{ALU2 Output}: is the output from the output flip-flops of ALU2. The pixel is clocked into the input of the Volume Adder on the next rising clock edge. This is also the final output which will be written into the memory selected to contain the output image.

\textit{START\_MEM(Y)}: originates from the Master Controller. It goes to the MEM.CONTROL chip controlling the memory selected to receive the final output of a processing operation. On the next rising clock edge, the memory address counter in the selected
MEM_CONTROL chip will start counting. The newly generated addresses (Y Addr) will be used by the memory selected to store the resultant pixels from ALU2.

**Y Addr:** is the address generated by the MEM_CONTROL chip controlling the memory designated to receive the resultant pixel from ALU2.

### 5.3.2 Overview of the MIP

The structural model of the MIP in figure 5.4 can be thought of as the realization of the MIP Data Path shown in figure 4.1. Although we have described in detail the MIP system and the individual components in Chapter 4, the inter-relations between the control units and the processing units were not discussed there. The purpose was to clearly illustrate the functionality of the MIP from the system point of view. However, it is important to see how the control units and the processing units work together to process an image as desired. Therefore, we will describe briefly the inter-relations between these units.

The MIP board is controlled by a host computer, which is emulated by the BLM model, PCBUS. All of the commands described in table 4.11 are from the host computer. Each command is associated with an address value and a data value. The address could indicate either a selected memory location or a register. If the address is for a selected memory location and the MEMWn or the MEMRn signals are activated, the corresponding Memory Controller will pass the address directly to the memory. If the address is for a register and the IOWn or the IORn signals are activated, the register address will be decoded by the Bus Interface to produce register control signals. These register control signals, denoted REGSn(13:0) from the Bus Interface, are connected to the Controller, the ALU1, the MAP, the ALU2, or the Volume Adder. The command from the host computer are sent to the registers in any of these components via SD bus.

A Memory Controller accepts control signals from both Bus Interface and Master Controller. When **PC-CS** is high, the control signals from the Bus Interface is for the host computer to access one of the on-board memories. When **PC-CS** is low, the control signals from the Controller is for processing units to access the on-board memories.

The memory bank provides input images for ALU1 or ALU2 via X1 Bus and X2 Bus. The output image of ALU1 is sent to MAP to be operated through either dilation or erosion. The window used by MAP for the dilation or erosion is down loaded from the host computer.
through SD Bus. The original and resultant images of MAP then enter ALU2. The third input image for ALU2 is from one of the memories. Two of the three input images will be manipulated by ALU2, and the processed image is sent back to the selected memory and to the Volume Adder. The Volume Adder will sum either the original or squared values of all pixels in the image and send the result back to the host computer via SD bus.

5.4 Simulation

The simulation for both the behavioral and the structural models of the MIP has been performed using a $32 \times 32$ image. The image file is in a ASCII format. Each pixel is represented by a four characters string in hexadecimal and delimited by a space. Each line contains 16 pixels. A $32 \times 32$ image was obtained through several steps. First, a graph was drawn by using Microsoft’s Painbrush. In order to obtain required image size, the graph was scanned through Xerox 7650 Scanner. The output from the scanner was in TIFF format, which was then converted into IMG format through a C program written by Yidong Chen. The IMG format is the display protocol used by PC to display the image. Finally, the IMG format was converted into ASCII format through another C program. A partial $32 \times 32$ image in ASCII format is shown in figure 5.6.

```
0100 0100 0100 0100 0100 0100 0100 0100 0100 0100 0100 0100 0100 0100 0100
0100 0100 0100 0100 0100 0100 0100 0100 0100 0100 0100 0100 0100 0100 0100
0100 0100 0100 0100 0100 0100 0100 0100 0100 0100 0100 0100 0100 0100 0100
0100 0100 0100 0100 0100 0100 0100 0100 0100 0100 0100 0100 0100 0100 0100
0100 0100 0100 00FF 001E 0022 0022 00FF 00FF 00FF 00FF 00FF 00FF 00FF 00FF
00FF 00FF 00FF 00FF 0017 0017 0017 001F 001F 005B 005B 00FF 00FF 00FF 0100
0100 0100 0100 0100 001E 001D 001E 0022 001F 00FF 00FF 00FF 00FF 00FF 00FF
00FF 00FF 00FF 00FF 0017 0017 0017 0017 001F 001F 001F 00FF 00FF 00FF 00FF
00FF 00FF 00FF 00DS 0047 0017 0017 0017 001F 001F 00FF 00FF 00FF 0100 0100 0100
```

Figure 5.6: A Partial $32 \times 32$ image in ASCII format

The identical output images were obtained from simulating both models by using the same input ASCII image. The resultant image was verified by running MIP.bin, a program written by Jens Rodenberg to exam the operations in MAP. Figure 5.7 illustrates the simulation result. The figure in top-left corner is the original image. The image was inverted for display purpose. Therefore, it should be realized that the dark area contains high-signal pixels while bright area has low-signal pixels. The original image is then eroded
consecutively by applying the window shown in table 5.2. The image in top-right corner is obtained after first erosion. The image size is smaller, and the low-signal area is expanded, i.e., the ring is wider. The images in lower-left and lower-right corners are obtained after second and third erosion, respectively.

Figure 5.7: Resultant Image, top-left: original image, top-right: first erosion, bottom right: second erosion, bottom left: third erosion

We noticed that simulation time for behavioral model is much shorter than that of structural model. For example, it took 16 minutes to complete simulation for a $32 \times 32$ image on Apollo 3500 workstation. The simulation time consumed by structural model depends on the size of the FIFO used. On the same workstation, it took 136 minutes if $32 \times 7$ FIFO is used, or 341 minutes if $512 \times 7$ FIFO is used. Assuming the simulation
time is linear with the array size, a 512 × 512 image will require 68 hours simulation time for a behavioral model, or 1455 hours (61 days) simulation time for a structural model! Therefore, it is impractical to simulate the structural model for a 512 × 512 image due to the simulation time required. However, it is sufficient to simulate the MIP based on 32 × 32 image since all the boundary conditions between 32 × 32 images and 512 × 512 images are identical. The major reason for slow simulation of structural model is due to its enormous number of signals, most of which require event scheduling during simulation. The simulation of a 512 × 512 natural image in Figure 5.8 was performed, and the resultant image is shown in Figure 5.9. We noticed first that the bottom part of the image is processed twice, but the top part of the image is disappeared. Secondly, the bottom part of the resultant image is black, i.e, the values of these pixels were not written to the resultant image file. The first observation is due to the memory access limitation of System 1076 version 7.0. Although it is not documented by Mentor Graphics, it seems that the maximum array size that can be achieved is 2^{17} words in integer type. The confusing point is that the system did not issue any error message when the array index exceeded the maximum array size. Instead, it overwrote the first half of the image. The second observation is due to the fact that our simulation time steps had exceeded the maximum simulation time steps of the system. In order to confirm the explanations given above, the first half of the image was simulated, the resultant image is shown on top of Figure 5.10. The bottom of Figure 5.10 is the processed image of the second half. It is evident that the simulation result is correct.

Table 5.2: The Window Array

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

62
Figure 5.8: Original 8-bit grey scale image
Figure 5.9: Resultant image of a complete $512 \times 512$ image after erosion
Since the BLM model of the PCBUS is to emulate the host computer, a command file for the PCBUS is required to run the simulation. We will explain in this section the command file listed below.

1  !map 32x32
2  WAP1
3  LINES 64
4  IOW 00030f 0044 #write enable+ Address = 200000h
5  IOW 00030e 0000 #memory load, select chip0 controller
6  IOW 00030d 0001 #PC -> memory0
7  PICLD 1fff fe images/co1.txt
Configure The Processor

8 IOW 00030e 0020
9 MASKLD 200000
10 00ff 00ff 00ff 00ff 00ff 00ff
11 00ff 00ff 00ff 00ff 00ff 00ff
12 00ff 00ff 0000 0000 0000 00ff 00ff
13 00ff 00ff 0000 0000 0000 00ff 00ff
14 00ff 00ff 00ff 00ff 00ff 00ff 00ff
15 00ff 00ff 00ff 00ff 00ff 00ff 00ff
16 00ff 00ff 00ff 00ff 00ff 00ff 00ff
17 #
18  # Setup to process m0 and output to m1
19 # ALU1: copy(A) ALU2: copy(B)
20 # map will do a MIN or erosion
21 #
22 !
23  ! Configure The Processor
24 !
25 IOW 000300 0000 # Sum X in Volume Adder
26 IOW 000305 0000 # alu1 const = 0
27 IOW 000306 0020 # alu1 op = copy(A)
28 IOW 000307 0000 # alu2 const = 0
29 IOW 000308 0020 # alu2 op = copy(A)
30 IOW 00030c 0000 # map mode = MIN
31 IOW 00030d 00f8 # M3: nop M2: nop M1: Y M0: X1
32 IOW 00030b 0000 # start the image
33 IOW 00030e 0008 # memory select chip1 controller
34 IOW 00030d 0002 # M1 -> PC
35 PICRD 1ffffe images/co2.txt
36 #
37  # read the Volume Adder
38 #
39 ! IOR 000300
40 IOR 000300
41 ! IOR 000301
42 IOR 000301
43 ! IOR 000302
44 IOR 000302
45 ! IOR 000303
46 IOR 000303
47 ! IOR 000304
48 IOR 000304
49 END 123456
50 END

Notice that # is a comment character. Anything after # is ignored. ! is an output text character, anything after ! is sent to the transcript window of Quicksim. WAP1 in line 2 is used to choose a 7 x 7 mask window and a 32 x 32 image size, while LINES 64 in line 3
instructs the model to expect 64 lines of 16 pixels each from the image file.

In line 4, Address Control Register (00030f) in Bus interface is configured so that write enable for either memory load or mask load is activated when the address is 200000 hex. Line 5 then sets up the Segment Register (00030e) by selecting Memory Controller 0 for memory load operation. The command in line 6 chooses Memory0 to store the original image from the host computer. Line 7 issues PICLD command so the image in host computer is loaded into selected memory, i.e., memory0 in this case. Since the PCBUS pre-increments the address, 1fffe instead of 200000 is used as the beginning address of the image stored in the host computer. After the image is loaded into memory0, the segment register is reconfigured for mask load. The values of the mask is from the command file between lines 10 and 16. It should be pointed out again that memory load must proceed mask load. Otherwise, the buffer between Y bus and the selected memory storing output image will be open and the output image will be lost.

The commands between line 25 and line 34 configure the processing units accordingly. Since copy(A) command is issued to both ALU1 and ALU2, the image is processed only in the MAP. For this particular example, the set up of constants for both ALU1 and ALU2 are irrelevant since they are not used in the process. In line 30, the erosion for the MAP is configured. Line 31 indicates that the original image is in memory0, and the resultant image will be memory1. Memory2 and memory3 are not used for the moment. Line 33 selects memory controller 1 so that the host computer can access the corresponding memory for reading the resultant image, which is accomplished by PICRD command in line 35. The last section reads the value stored in the register 000300 through register 000304 from the Volume Adder.
Chapter 6

Mathematic Units

The functionalities of the ALU1, the MAP, the ALU2 and the Volume Adder have been discussed in chapter 4. We will discuss in this chapter the implementation for these functions in VHDL. The operations of the four units are pipelined, i.e., the input image is shifted through the four stages pixel by pixel. Therefore, the speed of image processing is greatly improved.

Each of the operation units is described in three parts: a general description of the functionality, the input/output port description and the major processes in the architecture.

6.1 ALU1

The ALU1 is a pre-processor for the MAP. The two inputs of the ALU1 are connected with the on-board memory bank. The operations of the ALU are described in section 4.2.1. The output pixels are shifted into the MAP and the FIFO.

A unique ALU1 circuit is merely used in the FPGA version. In the VLSI version, the ALU1 is replaced by the ALU2 to reduce the manufacturing cost. This substitution was made because the functionality of the ALU1 is a subset of the functionality of the ALU2. Two of the operations (4x and 5x) provided by the ALU2 are not fully functional in ALU1. The operations 4x and 5x perform the “copy a” function.

6.1.1 Ports

The following list is the port description of the ALU1:

clk: is the on-board system clock.
a: connects with the memory bank through the X1 BUS as the first input of the ALU1.

b: connects with the memory bank through the X2 BUS as the second input of the ALU1.

start_alu1: indicates the start of a new image on the next rising edge of the clk and instructs the ALU1 to load the new operation and constant on the next clock.

sd: is the 8-bit input data bus for operations and constants. It is connected with bit (7 to 0) of the SD on the PC bus.

regsln: is an active low signal of register select status. The register address is hex 000305.

regshn: is an active low signal of register select status. The register address is hex 000306.

regehw: indicates write enable status. When it is active, a selected register will latch its new contents from SD bus.

alu1_out is the output of the ALU1.

6.1.2 Processes

The processes alu1_a_read_and_convert and alu1_b_read_and_convert are used to convert the bus input into an integer. We will not discuss the processes further since the similar code has already discussed in section 5.2.2.

The process load_op_to_buffer latches the new value on the SD bus, sd, to register aluop_buf, when its address is selected and the write enable, regehw, is active.

```
180  load_op_to_buffer:
181  PROCESS
182  BEGIN
183      WAIT ON regehw, regsln UNTIL (regehw = '1') AND (regsln = '0');
184      aluop_buf <= sd;
185  END PROCESS load_op_to_buffer;
```

The process load_const_to_buffer latches the new value on the sd to register aluconst_buf, when its address is selected and the write enable is active.

```
186  load_const_to_buffer:
187  PROCESS
188  BEGIN
189      WAIT ON regehw, regsln UNTIL (regehw = '1') AND (regsln = '0');
190      aluconst_buf <= sd;
191  END PROCESS load_const_to_buffer;
```
The process load_new_const_and_op latches the new constant and operation value from the register aluop_buf and aluconst_buf to the register aluop and aluconst when the start_alul is active. The aluop defines the operation of the ALU1 while the aluconst stores the constant value used in some operations.

```vhdl
192   load_new_const_and_op:
193     PROCESS
194       VARIABLE temp,i : integer;
195     BEGIN
196       WAIT ON clk UNTIL (clk = '1') AND (start_alul = '1');
197       temp:=0;
198       FOR i IN 7 DOWNTO 4 LOOP
199         CASE (aluop_buf(i)) IS
200           WHEN '1' =>
201             temp := temp*2+1;
202           WHEN '0' =>
203             temp := temp*2;
204           WHEN 'X'|'Z' =>
205             temp := UNKNOWN;
206             EXIT;
207         END CASE;
208       END LOOP;
209       aluop <= temp;
210       temp:=0;
211       FOR i IN 7 DOWNTO 0 LOOP
212         CASE (aluconst_buf(i)) IS
213           ... same code as 199-207...
214         END CASE;
215       END LOOP;
216       IF (aluop_buf(0) = '1') AND (temp /= UNKNOWN) THEN
217         temp := temp - 256;
218       END IF;
219       aluconst <= temp;
220     END PROCESS load_new_const_and_op;
```

The process alul_processing models a combinational circuit within the ALU1. Whenever the operation (aluop), constant (aluconst), input a (alul_a), or input b (alul_b) is changed, the process will re-evaluate the output value (output).

```vhdl
227   alu1_processing:
228     PROCESS
229       VARIABLE op,const,in_a,in_b : integer;
230     BEGIN
231       WAIT ON aluop,aluconst,alul_a,alul_b;
232       op := aluop;
```
const := aluconst;
in_a := alu1_a;
in_b := alu1_b;
output <= alu1(in_a, in_b, const, op);
END PROCESS alu1_processing;

The function alu1 used in alu1_processing includes all of the mathematic functions of the ALU1.

FUNCTION alu1(a, b, aluconst, aluop: integer)
    RETURN integer IS
    VARIABLE output: integer := UNKNOWN;
    BEGIN
        CASE (aluop) IS
            WHEN MIN1AB | MIN2AB =>
                IF (compare(a, b, MIN) = TRUE) THEN
                    output := b;
                ELSE
                    output := a;
                END IF;
            WHEN MAX1AB | MAX2AB =>
                IF (compare(a, b, MAX) = TRUE) THEN
                    output := b;
                ELSE
                    output := a;
                END IF;
            WHEN COPY1A | COPY2A =>
                output := a;
            WHEN COPY1B | COPY2B =>
                output := b;
            WHEN ADDAB1 | ADDAB2 =>
                output := adder(a, b);
            WHEN ADDAC1 | ADDAC2 =>
                output := adder(a, aluconst);
            WHEN SUBAB1 =>
                IF (b = MFIN) THEN
                    output := MFIN;
                ELSE
                    output := adder(a, -1*b);
                END IF;
            WHEN SUBAB2 =>
                IF (b = MFIN) THEN
                    output := a;
                ELSE
                    output := adder(a, -1*b);
                END IF;
            WHEN SUBAC1 =>


IF (aluconst = MFIN) THEN
    output := MFIN;
ELSE
    output := adder(a, -1*aluconst);
END IF;
WHEN SUBAC2 =>
    IF (aluconst = MFIN) THEN
        output := a;
    ELSE
        output := adder(a, -1*aluconst);
    END IF;
WHEN OTHERS =>
    output := UNKNOWN;
END CASE;
RETURN output;
END alul;

It should be realized that the function alul is based on comparison and addition. These two operations are also implemented with functions. The function compare returns either 0 or 1 to indicate the relation between the arguments a and b.

FUNCTION compare(a,b,max : integer)
    RETURN integer IS
    VARIABLE comp : integer;
    BEGIN
    IF ((a > b) XOR (max = 0)) THEN
        comp := FALSE;
    ELSE
        comp := TRUE;
    END IF;
    RETURN comp;
END compare;

The adder function adds two operands and returns the summation. It should be noticed that no overflow or underflow flag existed in the circuit design. If this situation happens, the result will be either maximum or minimum.

FUNCTION adder (a,b : integer)
    RETURN integer IS
    VARIABLE sum : integer;
    BEGIN
    IF ((a = UNKNOWN) OR (b = UNKNOWN)) THEN
        sum := UNKNOWN;
    ELSIF ((a = MFIN) OR (b = MFIN)) THEN
        sum := MFIN;
    ELSE
        sum := adder(a, b);
    END IF;
    RETURN sum;
END adder;
ELSIF ((a >= 0) AND (b >= 0) AND ((a + b) > MAXNUM)) THEN
sum := MAXNUM;
ELSIF ((a < 0) AND (b < 0) AND ((a + b) < MINNUM)) THEN
sum := MINNUM;
ELSE
sum := a + b;
END IF;
RETURN sum;
END adder;

The output value, alu_out, is converted from an integer, output, to a bus vector in the output_processing.

output_processing:
PROCESS
VARIABLE temp,i,choice : integer;
BEGIN
WAIT ON clk UNTIL clk='1';
temp := output;
IF (temp < 0) THEN
temp := temp + 512;
END IF;
IF (output /= UNKNOWN) THEN
FOR i IN 0 TO 8 LOOP
choice := temp mod 2;
CASE (choice) IS
WHEN 1 =>
alu_out(i) <= TRANSPORT '1' AFTER DELAY;
WHEN 0 =>
alu_out(i) <= TRANSPORT '0' AFTER DELAY;
WHEN OTHERS =>
NULL;
END CASE;
temp := temp / 2;
END LOOP;
ELSE
FOR i IN 0 TO 8 LOOP
alu_out(i) <= TRANSPORT 'X' AFTER DELAY;
END LOOP;
END IF;
END PROCESS output_processing;
6.2 MAP

The MAP accepts an input image from the ALU1, performs the morphological operations, and outputs the processed image as well as the original image to the ALU2. The architecture of the MAP is designed to perform two basic morphological operations: dilation and erosion. The algorithm used in this section is different from the algorithm inserted in section 2.4. The dilation algorithm in section 2.4 processes an image by sliding a window through the image. During processing, the central pixel of the window is aligned with a target pixel in the image; then, the additions between the window and the pixels surrounding the target pixel are performed; next, the target pixel value is replaced by the maximum value of the summations; the procedures are repeated through every pixel of the image. The erosion is similar to the dilation except a negated and rotated window is used and the minimum value of the summations is chosen as the resultant value.

In the pipelined architecture designed by Jens Rodenberg and Jeff Hanzlik, a pixel processed by the ALU1 proceeds to the MAP, the ALU2 and the Volume Adder. The algorithm in section 2.4 requires a moving window and a stored complete image which cannot be provided during the pipelined process. In addition, it requires a buffer to store the original image which has to be pre-loaded before the operation starts. Therefore, it is necessary to modify the algorithm for a pipelined architecture. The new algorithm will use a still window and a moving image. It also reduces the size of the buffer, for example, from $512 \times 512$ to $512 \times 7$. The modified algorithm is presented in this section.

The architecture has been partitioned into FIFO and MAP. They are separated for two reasons. First of all, it lowers the development cost because FIFO chips are commercially available. Secondly, it is more flexible since the column width of the array can be changed to match the image’s column width. The column width of an image could be either 512 or 1024 in the VLSI version, although it is fixed at 512 in the FPGA version.

The FIFO is a shift register array. The size of the array is either 7 rows $\times$ 512 columns or 7 rows $\times$ 1024 columns, depending on the image size. The input of the first pixel is connected with the output of the ALU1. Therefore, each output pixel from the ALU1 is shifted into this shift register array. The shift register array functions as a line delay to store temporarily 7 rows of the image.

The MAP consists five parts: the image buffer block, the window buffer block, the
window register block, the adder block and the comparison tree. The image buffer block consists of 7 shift register arrays. Each array has 7 registers. The input pixel of each array is the first pixel of the corresponding row of the FIFO. With this connection, the image is shifted through the image buffer row by row. The center of the image buffer is the target pixel of the morphological operations. The window buffer is a $7 \times 7$ shifter register array. The Input of the array is connected with the SD bus. A new window value is shifted into the array upon the rising edge of the w clk. The contents of the window buffer is latched by the window registers on the rising edge of the start proc as the new window for the morphological operations. The adder block adds the window registers with the image buffer to produces 49 summations. The summations are compared by the comparison tree to find out the maximum or minimum value as the result for the target pixel.

For the target pixels located within the first and last three rows as well as the first and last three columns of an image, part of the surrounding pixels used in the addition are outside of the image boundary. Therefore, the $-\infty$ value is used for summation value to indicate that the pixel value is undefined. The controlling mechanism is provided by the Master Controller which generates the row and column blanking signals through the blank counter. Table 6.1 written by Jens Rodenberg shows the blanking sequence. The row blanking bits 0 to 5 and the column blanking bits 0 to 5 are connected to the adder blocks in the MAP. The bits 0 to 2 of the row blanking bits indicates the blanking status of row 0 to 2 while the bits 3 to 5 indicates the status of row 4 to 7. The connecting sequence is the same in the column blanking bits.

A 9-bit signed integer is used to represent a pixel. Values 0 to 255 are used to indicate the grey scale levels for the pixel. Values -1 to -255 are used for a negated pixel value in the window for erosion. The $-\infty$ is coded as -256 for an undefined pixel value.

The MAP has been through three different implementations. After the architecture was designed, Jeff and Jens realized that the circuit was much too large to fit into one FPGA chip. They partitioned the entire circuit into 24 FPGA chips. Each of the chips contains part of the addition block and the comparison block. This affects the architecture by extending the pipelined stages from 7 to 14. When Larry Rubin implemented the architecture in an ASIC, he packed the whole MAP into one ASIC chip. However, the manufacturing cost for the single chip is quit expensive. In order to control the cost to a reasonable range, Chris Insalaco and Shishir Ghate partitioned the MAP into 7 ASIC
Figure 6.1: The Blanking Sequence of the MAP
chips. To accommodate various implementations of the MAP and future modifications, a universal VHDL model with variable output delay is designed. The VHDL models for the partial circuit of the MAP, (CHIP A, CHIP B, CHIP C and XFIFO) in FPGA version are included in the appendix for further reference.

6.2.1 Ports

The following list is the port description of the MAP.

clk : is the on-board system clock.

w_clk : is the clock for the window buffer. The new window value is shifted in by this clock. It provides the ability for the system to load a new set of window values to window buffers when the current MIP process is still running.

w : is the input for a new window value. It is connected with the SD bus.

x0 to x6 : are the row inputs for the image buffer.

rowblnk : is generated by the Master Controller to indicate the row blank status.

colblnk : is generated by the Master Controller to indicate the column blank status.

max : indicates the operation of the MAP. Max is '1' for dilation and '0' for erosion.

start_proc : instructs the MAP to start processing by latching the new window and operations on the rising edge of the next clock. Also, the target pixel on the next clock is the first valid pixel of the new image.

y : is an output port for a pixel from the resultant image.

xo : is an output port for a pixel from the original image.

6.2.2 Processes

Several constants in the architecture are used to specify the pipelined stages.

134 CONSTANT ROW : integer := 7;
136 CONSTANT OPTION : integer := 7; -- 7 more delay for jandj
137 CONSTANT YDELAY : integer := 5+OPTION;
The ROW constant declares the row size of the window. The YDELAY is the pipelined stages excluding the delay due to the input and output stages. The OPTION is the extra pipelined stages introduced by splitting the MAP into different chips.

The first eight processes in the model are used to convert the bus value into integer. They are not discussed since the similar code has been shown in the ALU1 process, load_new_const_and_op.

The rowblnk_read_and_convert process converts the 6-bit vector into a 7-bit vector. The resultant vector is more compatible with the $7 \times 7$ window.

```vhdl
map_7x7_rowblnk_read_and_convert:
PROCESS
  VARIABLE i: integer;
BEGIN
  WAIT ON rowblnk;
  FOR i IN 0 TO 2 LOOP
    CASE (rowblnk(i)) IS
      WHEN '1' =>
        rowb(i) <= SET;
      WHEN '0' =>
        rowb(i) <= RESET;
      WHEN 'X'|'Z' =>
        rowb(i) <= UNKNOWN;
    END CASE;
  END LOOP;
  rowb(3) <= RESET;
  FOR i IN 3 TO 5 LOOP
    CASE (rowblnk(i)) IS
      WHEN '1' =>
        rowb(i+1) <= SET;
      WHEN '0' =>
        rowb(i+1) <= RESET;
      WHEN 'X'|'Z' =>
        rowb(i+1) <= UNKNOWN;
    END CASE;
  END LOOP;
END PROCESS map_7x7_rowblnk_read_and_convert;
```

When the w_clk is active, the window buffer array called wreg is shifted in one word. Because the wreg is a signal array which can not be re-evaluate in the same delta interval, a variable array, temp_w, is used for shifting the values. The variable is then copied back to the original signal array.

```vhdl
shift_w_process:
```
PROCESS
VARIABLE temp_w : window_array;
BEGIN
WAIT ON neww,w_clk UNTIL w_clk='1';
FOR i IN SIZE-1 DOWNTO 1 LOOP
  temp_w(i) := wreg(i-1);
END LOOP;
temp_w(0) := neww;
wreg <= temp_w;
END PROCESS shift_w_process;

The clk_process is synchronized with the system clock. Every clock-related statement is placed in this process. Since variables are used for signal propagation, it should be noted that the evaluation result of a variable can be accessed by the latter statement in the same delta interval. The right sequence of the statements will keep the variable from being accessed until next clock. For instance, if a variable is the output in statement A and the input for the statement B, the statement B must be placed before the statement A. Otherwise, the change of the statement A will be reached incorrectly to the output of the statement B in the same clock period.

The delayx and delayy are two output buffers used to perform the output delay. The number of these delay stages are defined by constants XDELAY and YDELAY.

clk_process:
PROCESS
  ... variable declaration deleted
BEGIN
  WAIT ON startp,clk UNTIL clk = '1';
  SHIFT X0
  FOR i IN XDELAY-1 DOWNTO 1 LOOP
    delayx(i) := delayx(i-1);
  END LOOP;
  delayx(0) := xreg((SIZE-1)/2);
  SHIFT Y0
  FOR i IN YDELAY-1 DOWNTO 1 LOOP
    delayy(i) := delayy(i-1);
  END LOOP;
  delayy(0) := newy;

The statements shown next are to find the maximum or minimum value of the summation according to the value of maxreg, the MIP operation register. The function compare has been discussed in the previous section.
337-- COMPARE BLOCK
338    temp_cmp := sum(0);
339    FOR i IN SIZE-1 DOWNTO 0 LOOP
340        IF (compare(temp_cmp,sum(i),maxreg) = TRUE) THEN
341            temp_cmp := sum(i);
342        END IF;
343    END LOOP;
344    newy <= temp_cmp;

The image pixels and the window registers are added in the following statement. The column and row blanking status are taken care here.

346-- ADDER BLOCK
347    FOR i IN ROW-1 DOWNTO 0 LOOP
348        FOR j IN ROW-1 DOWNTO 0 LOOP
349            temp_sum(i*ROW+j) := adder(xreg(i*ROW+j),wlreg(i*ROW+j),
                                           rowb(i),colb(j));
350        END LOOP;
351    END LOOP;
352    sum <= temp_sum;

The image pixels stored in the FIFO are then shifted in every clock from the register array newx whose values are the integers converted from the bit vectors of the ports x0 to x6. These pixels are shifted through the buffer xreg.

354-- SHIFT X
355    FOR i IN ROW-1 DOWNTO 0 LOOP
356        FOR j IN ROW-1 DOWNTO 1 LOOP
357            temp_x(i*ROW+j) := xreg(i*ROW+j-1);
358        END LOOP;
359    END LOOP;
360    END LOOP;
361    xreg <= temp_x;

If the start_proc is active, the new value of the window registers and the MAP operation are latched on the rising edge of the next clock.

364    IF (startp = SET) THEN
365        maxreg <= newmax;
366        wlreg <= wreg;
367    END IF;

The integer outputs are converted to type qsim_state_vector for the output ports.
368  out_gen(delayx(XDELAY-1),xotemp);
369  out_gen(delayy(YDELAY-1),yotemp);
370
371  xo <= TRANSPORT xotemp AFTER DELAY;
372  y  <= TRANSPORT yotemp  AFTER DELAY;
373  END PROCESS clk_process;

The function adder simulates the adders used in the MAP. The additional arguments are
the rb for row blanking and the cb for column blanking status. If either rb or cb is set, the
output is $-\infty$.

38 FUNCTION adder (a,b,rb,cb : integer)
39   -- rb: state of row blanking signal
40   -- cb: state of column blanking signal
41   RETURN integer IS
42     VARIABLE sum : integer;
43   BEGIN
44     IF ((a = UNKNOWN) OR (b = UNKNOWN)
45         OR (rb = UNKNOWN) OR (cb = UNKNOWN)) THEN
46       sum := UNKNOWN;
47     ELSIF ((a = MFIN) OR (b = MFIN)
48         OR (rb = SET) OR (cb = SET)) THEN
49       sum := MFIN;
50     ELSIF ((a >= 0) AND (b >= 0) AND ((a + b) > MAXNUM)) THEN
51       sum := MAXNUM;
52     ELSIF ((a < 0) AND (b < 0) AND ((a + b) < MINNUM)) THEN
53       sum := MINNUM;
54     ELSE
55       sum := a + b;
56     END IF;
57     RETURN sum;
58   END adder;

6.3 ALU2

The ALU2 has three external inputs: one is the pixel value from the resultant image of the
MAP, one is the pixel value from the original image from the MAP, and one is the pixel
value from an on-board memory bank. Only two of the three inputs are connected to the
internal inputs. The functionality of the ALU2 is similar to the ALU1 with two additional
operations: to find the maximum or minimum pixel value of an image. The output of the
ALU2 goes to the Volume Adder and a selected on-board memory.
6.3.1 Ports

The following are the input and output ports of the ALU2:

**clk**: is the on-board system clock.

**a**: connects to one of the MAP's output port, \( y \), which is the pixel of the resultant image from the morphological operations.

**b**: connects to one of the MAP's output port, \( xo \), which is the target pixel of the morphological operations.

**c**: is the pixel from the on-board memory bank through X2 BUS.

**start_alu2**: indicates the start of a new image on the next clock period and instructs the ALU2 to load the new operation and constant.

**stop_alu2**: indicates the end of an image and instructs the ALU2 to latch the maximum/minimum search result.

**sd_in**: is the input data bus for operations and constants.

**sd_out**: is the output data bus for maximum/minimum output registers, 000307 and 000308. The register 000307 stores the bit 7 to 0 of the maximum/minimum value and the register 000308 stores the bit 8 of the value.

**regsln**: is an active low signal of the register-select status for the register 000307.

**regshn**: is an active low signal of the register-select status for the register 000308.

**regeuw**: is an active high signal for the register write enable.

**regewr**: is an active high signal for the register read enable.

**alu2_out** is the pixel output of the ALU2.

6.3.2 Processes

Most of the processes in the ALU2 are identical with that in the ALU1. These processes have been discussed in the section, ALU1. The processes which are unique to the ALU2 will be discussed in this section. One of the differences between ALU1 and ALU2 is that the
data bus is uni-direction for the ALU1 and bi-direction for the ALU2. Since the system 1076 version 7.0 does not support INOUT type, the bi-direction bus is handled by the process sd_handle which connects the sd.in and sd.out inside the process in order to emulate the INOUT port type.

Another difference is that the ALU1 has two inputs, but the ALU2 has three inputs. Evidently, only two of the three inputs of ALU2 can be used as operands. The selection is controlled by the register xbusmode which maps the value of the bits 2 and 3 of the register 000308. The selected inputs are copied into the internal registers, aff and bff, as operands.

load_new_aff_and_bff:

PROCESS
BEGIN
WAIT ON clk, xbusmode UNTIL clk='1';
CASE xbusmode IS
WHEN NORM =>
aff <= newa;
bff <= newb;

WHEN CTOA =>
aff <= newc;
bff <= newb;

WHEN CTOB =>
aff <= newa;
bff <= newc;

WHEN CAB =>
aff <= newc;
bff <= newc;

WHEN OTHERS =>

END PROCESS load_new_aff_and_bff;
The ALU2 supports the search capability of the maximum or the minimum pixel in an image. The following code supports the functionality. For the maximum search, the register called loopback stores the current maximum value. If the input pixel value is larger than the value in the loopback, the value in the loopback will be replaced by the input a. In addition, the a is copied to the output register outff, so the output image is a copy of the input image. The procedures in searching minimum is identical with that for the maximum. The register loopback, however, will contain the minimum value instead of the maximum value of the image.

The search result from the loopback is latched to the register regout when the stop_alu2 signal is active. The regout is the combination of the output register 000307 and 000308.

The output register 000307 or 000308 is read out when the address is selected and the read enable is active. The output register, regout, stores the maximum/minimum value. When the address 000307 is selected (regsln='0'), the bit 7 to 0 is read out through sd_out.
When the address 000308 is select( regshn='0'), the bit 8 is read out. When the read enable is inactive, the sd_out value is Z as hi-impedance.

337    regenr_process:
338    PROCESS(regenr)
339        VARIABLE temp, i, choice : integer;
340    BEGIN
341        -- process the 0 - 7 bit of regout
342        IF (regenr = '1') AND (regshn = '0') THEN
343            temp := regout;
344            IF (temp < 0) THEN
345                temp := temp + 512;
346            END IF;
347            temp := temp mod 255; -- regout & 0xff
348            FOR i IN 0 TO 7 LOOP
349                choice := temp mod 2;
350                temp := temp / 2;
351                CASE (choice) IS
352                    WHEN 1 =>
353                        sd_buf(i) <= TRANSPORT '1' AFTER DELAY;
354                    WHEN 0 =>
355                        sd_buf(i) <= TRANSPORT '0' AFTER DELAY;
356                    WHEN OTHERS =>
357                        NULL;
358                END CASE;
359            END LOOP;
360        -- process the 8th bit of regout
361        ELSIF (regenr = '1') AND (regshn = '0') THEN
362            FOR i IN 1 TO 7 LOOP
363                sd_buf(i) <= TRANSPORT '0' AFTER DELAY;
364            END LOOP;
365            temp := regout;
366            IF (temp < 0) THEN
367                temp := temp + 512;
368            END IF;
369            IF (((temp / 2**8) mod 2) = 1) THEN
370                sd_buf(0) <= TRANSPORT '1' AFTER DELAY;
371            ELSE
372                sd_buf(0) <= TRANSPORT '0' AFTER DELAY;
373            END IF;
374        END IF;
375        -- no operation
376        ELSIF (regenr = '0') THEN
377            FOR i IN 0 TO 7 LOOP
378                sd_buf(i) <= TRANSPORT 'Z' AFTER DELAY;
379            END LOOP;
380        END IF;
381    END PROCESS regenr_process;
6.4 Volume Adder

The Volume Adder accumulates either the absolute value or the squared value of the output pixels from the output of the ALU2 and stores the result in registers. The squared values of all 8-bit integer are calculated ahead and stored in ROM as a look-up table.

6.4.1 Ports

The following list is the port description of the Volume Adder.

clk : is the on-board system clock.

x : connects with the output of the ALU2.

xs : connects with the squared output of the ALU2.

start_alu2 : indicates that the first valid pixel will arrive after two clock periods. It instructs the Volume Adder to reset, then starts accumulating the input values.

stop_alu2 : indicates that the last valid pixel will arrive after three clock periods. It instructs the Volume Adder to stop adding and stores the volume to the output registers.

rengenw : is the register write enable signal. When it is active, and the bit 0 of the regsn is low, the Volume Adder will latch the new operation value from sd_in.

regenre : is the register read enable signal. When it is active, the contents of the address selected by regsn is sent to sd_out.

regrsn : is the active low signal for address selection. The bit 0 to bit 4 is used to select the register 000300 to 000304.

sd_in : is the input port from the SD bus.

sd_out : is the output port to the SD bus.

6.4.2 Processes

The regenw_process is to latch the new Volume Adder operation to the buffer mode_1.
186 regenw_process:
187 PROCESS
188 BEGIN
189 WAIT ON newsd, regsnreg, regenw UNTIL regenw='1' AND regsnreg=REGO;
190 mode_1 <= newsd mod 2; -- newsd & 0x01
191 END PROCESS regenw_process;

The regenr_process is to read out one of the registers, REG0 to REG5 whose hexadecimal address 000300 to 000304, to the SD bus.

193 regenr_process:
194 PROCESS
195 VARIABLE temp_sd : qsim_state_vector(7 DOWNTO 0);
196 BEGIN
197 WAIT ON regsnreg, regenr UNTIL regenr='1';
198 CASE (regsnreg) IS
199 WHEN REG0 =>
200 out_gen(out0,temp_sd);
201 WHEN REG1 =>
202 out_gen(out1,temp_sd);
203 WHEN REG2 =>
204 out_gen(out2,temp_sd);
205 WHEN REG3 =>
206 out_gen(out3,temp_sd);
207 WHEN REG4 =>
208 out_gen(out4,temp_sd);
209 WHEN OTHERS => NULL;
210 END CASE;
211 sd_buf <= TRANSPORT temp_sd AFTER DELAY3;
212 END PROCESS regenr_process;

The values of the start_alu2 and stop_alu2 are buffered into variables start and stop whenever they change.

213 clk_process:
214 PROCESS
215 ... variable declaration deleted
216 BEGIN
217 WAIT ON start_alu2,stop_alu2,clk;
218 IF (start_alu2 = '0') THEN
219 start := RESET;
220 ELSE
221 start := SET;
222 END IF;
223 IF (stop_alu2 = '0') THEN
224 stop := RESET;
ELSE
    stop := SET;
END IF;

When the variable stop2 equals SET, it indicates that the last valid pixel is done and that the result is partitioned into 8-bit register format and stored in the output registers, out0 to out4.

IF (clk = '1') THEN
    IF (stop2 = SET) THEN
        out0 <= sum MOD 16#100#;  --(sum & 0xff)
        out1 <= (sum/2**8) MOD 16#100#;  --(sum & 0xff00)>>8
        out2 <= cnt18 MOD 16#100#;  --(cnt18 & 0xff)
        out3 <= (cnt18/2**8) MOD 16#100#;  --(cnt18 & 0xff00)>>8
        temp_out4 := ((cnt18/2**17) MOD 2) * 2;  --(cnt18 & 0x20000)>>16
        IF (sign = SET) THEN
            temp_out4 := temp_out4 + 16#80#;  --out4 |= 0x80
        END IF;
        out4 <= temp_out4;
    END IF;
END IF;

When the variable start1 equals SET, it instructs the Volume Adder reset the internal registers. Otherwise, the Volume Adder keeps adding the input value every clock. If the summation is more then 16 bits, the 18-bit counter is used to store the rest bits.

IF (start1 = SET) THEN
    sum <= addin;
    cnt18 <= 0;
    sign <= RESET;
    mode_sel <= mode_1;
ELSE
    temp_sum := adder(sum,addin);
    sum <= temp_sum MOD 16#10000#;
    IF (temp_sum > 16#ffff#) THEN
        cnt18 <= cnt18+1;
    END IF;
END IF;

During every clock period, the start value is shifted through two flip-flops. It creates a two clock period delay since it takes two clock period for the ALU2 to generate the first valid pixel. The stop value goes through three flip-flops to indicate the last valid pixel is done.
start0_temp :=start0;
stop1_temp :=stop1;
stop0_temp :=stop0;
start1 <= start0_temp;
start0 <= start;
stop2 <= stop1_temp;
stop1 <= stop0_temp;
stop0 <= stop;

The mode_sel variable chooses the input from either the squared input, newxs, or the absolute value from the newx.

IF (mode_sel = SET) THEN
  addin <= newxs;
ELSE
  addin <= newx MOD 16#100#;
END IF;

The variable sign is a flag which shows that a negative value is detected from the input.

IF (sign /= SET) AND (sign0 = SET) THEN
  sign <= SET;
END IF;

IF ((newx /2**8)= 1) THEN
  sign0 <= SET;
ELSE
  sign0 <= RESET;
END IF;
END PROCESS clk_process;

The simulation of all models are performed with Mentor Graphics’ Quicksim. The do file of the ALU1 and ALU2 are provided by Jeff Hanzlik. The results matches the specification.
Chapter 7

Memory Units

The on-board memory chip model and the tri-state buffers on the local buses will be described in this chapter.

7.1 Memory

This memory model is a general read-only memory behavioral model. A more accurate model should be provided by the manufacturer for a specific commercial chip in a future timing model of the system.

7.1.1 Ports

D: is the input of new contents.

Q: is the output of the contents.

ADDR: is the selected address.

WEn: is an active-low write enable signal.

CSn: is an active-low chip select.

7.1.2 Process

The chip must be selected for either read or write (CSn='0'). Then, if the WEn='0', the memory updates the contents on the address selected by ADDR. During the write enable mode, the output value of the memory is UNKNOWN because the value on a real circuit
is unstable during writing in. If the WEn is '1', the chip is read enable. The contents on the address is output to the port Q.

```vhdl
memory_main_process:
PROCESS
BEGIN
VARIABLE memory: memory_type;
BEGIN
wait on D_REG, ADDR_REG, WEn, CSn;
if (CSn='0') then
  if ( (ADDR_REG<0) or (ADDR_REG>(RAM_SIZE-1)) ) then
  assert FALSE
  report "INVALID MEMORY ADDRESS"
  severity WARNING;
  else
  case (WEn) is
  when '0' => memory(ADDR_REG) := D_REG; -- INPUT
  Q_REG <= transport UNKNOWN after MEMR_DELAY;
  -- OUTPUT UNKNOWN
  when '1' => Q_REG <= transport memory(ADDR_REG)
  after MEMR_DELAY;

  -- OUTPUT
  when others =>
  assert FALSE
  report "INVALID WEn VALUE"
  severity WARNING;
  end case;
end if;
end if;
END PROCESS memory_main_process;

END memory_behavior;
```

7.2 Buffer

This is a tri-state buffer. The output of the buffer is controlled by the enable pin on the buffer: when enable pin is active (low), the buffer is turned on and the input signal passes through the buffer to the output pin; otherwise, the buffer is turned off and the output of the buffer is hi-impedance.

7.2.1 Ports

The following list is the port description of the buffer.

BUFF_IN: is the input of the buffer.
**BUF.OUT:** is the output of the buffer.

**En:** En is the active-low buffer enable signal.

### 7.2.2 Process

The BUF.IN can only pass to the BUF.OUT when the enable is active.

```
37 BEGIN
38 with En select
39   BUF.OUT <= high_impedence when '1',
40   BUF.IN when '0',
41   buf_unknown when others;
42 END buffer_behavior;
```

### 7.2.3 Further Implementation

The table 7.1 is a four state resolve table for the tri-state signals. Since the signal resolving is not part of system 1076 version 7.0, the actual signal is resolved by a BLM in the same resolving table.

<table>
<thead>
<tr>
<th>IN2</th>
<th>Z  0 1 X</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>Z Z 0 1 X</td>
</tr>
<tr>
<td>N</td>
<td>0 0 0 X X</td>
</tr>
<tr>
<td>1</td>
<td>1 1 X 1 X</td>
</tr>
<tr>
<td></td>
<td>X X X X X</td>
</tr>
</tbody>
</table>

Table 7.1: Tri-state Signal Resolving Table

The following data type and code can be used for a tri-state bus signal in a full-implemented VHDL system:

```vhdl
TYPE tri_bus IS ARRAY (INTEGER range <>) OF qsim_state_vector;
FUNCTION bus_resolve(unsolved_bus_vectors: tri_bus)
return qsim_state_vector is
  VARIABLE temp_vector: qsim_state_vector;
  VARIABLE case_vector: qsim_state_vector(1 downto 0);
begi
  temp_vector := unsolved_bus_vectors(unsolved_bus_vector'LOW);
  for i in unsolved_bus_vectors'_RANGE loop
    for j in unsolved_bus_vectors(i)'_RANGE loop
      case_vector := temp_vector(j)&unsolved_bus_vector(j);
  ```
case case_vector is
  when "ZZ"|"Z0"|"Z1"|"ZX" =>
    temp_vector(j) := unsolved_bus_vectors(i)(j);
  when "00"|"0Z" =>
    temp_vector(j) := '0';
  when "11"|"1Z" =>
    temp_vector(j) := '1';
  when others =>
    temp_vector(j) := 'X';
end case;
end loop;
end loop;
return temp_vector;
end bus_resolve;
SUBTYPE res_bus_vector is bus_resolve qsim_state_vector;
Chapter 8

Conclusion

In this project, the theory of the MIP has been studied, and the behavioral and structural models of the MIP have been established as well as simulated in VHDL. The behavioral model can be used for future system development since it incorporates the functionality as well as the timing information of the MIP from the design specifications.

The structural model, on the other hand, can be used to document the designed MIP system. A shared BLM testbench has been used for both models, and the simulation results are identical for a $32 \times 32$ image. Although the MIP is designed for $512 \times 512$ image processing, it is sufficient to simulate the MIP based on a $32 \times 32$ image since the boundary conditions between the two are the same. It is clearly an advantage to use a behavioral VHDL model when simulating the MIP since it is impractical to simulate the MIP at a gate level.

The MIP system is partitioned into separate functional blocks: I/O unit, Control Units, Arithmetic Units, and Memory Units. While the behavioral model is based on the functionality of the MIP, the structural model is based on the partitioned functional blocks. Each functional block shown in figure 5.3 is composed of one or more VHDL models corresponding to their physical blocks. Since the objective of this project was to provide the documentation for the MIP system, the VHDL models are constructed only for the physical blocks in the system.

Although the mission of documenting the MIP system has been accomplished with this project, it is possible for interested readers to further extend the project. One possibility is to create a VHDL library for logic elements at gate level and use these basic models to
construct the structural models of the physical blocks currently represented by the behavioral models. Another possibility is to create a new MIP using synthesis tool based on the current structural model to further decompose the structural models into finer functional blocks.
Bibliography


Appendix A

Bus Interface

We will discuss the Bus Interface between the host computer and the MIP board in this chapter. The Bus Interface accepts commands from the host computer and generates the corresponding register control and read/write signals for the various on-board components. Before discussing the implementation of the circuit modeling in VHDL, we will describe the input and output signals of the Bus Interface.

A.1 Input and Output Signals

It should be realized that a signal name without the letter "n" at the end indicates that the signal is active high, while a signal with "n" at the end indicates that the signal is active low.

RESET: is used to generate a power-on reset to bring the MIP to a known state before its operation.

BCLK: is an 8-MHz bus clock from the host computer. The clock is used to synchronize the output signals generated by the Bus Interface.

BALE: updates address information for the Bus Interface when it is high.

SA(15:0): generates register control signals REGSn and register read/write signals through the IO.Decoder in the Bus Interface.

LA(23:19): are used to provide memory address information about the present bus cycle. The information is latched on the falling edge of BALE signal.
**SD(7:0):** contains information to generate the control signals.

**MEMWn:** generates the control signals for the data transfer from the system bus to onboard memory.

**MEMRn:** generates the control signals for the data transfer from the on-board memory to the host computer.

**IOWN:** generates the control signals for the data transfer from the system bus to an I/O register.

**IORn:** generates the control signals for the data transfer from an I/O register to the host computer.

**SW(1:0):** configures the addresses for I/O registers. Table A.1 shows the correspondence between **SW(1:0)** and addresses for I/O registers.

<table>
<thead>
<tr>
<th>SW(1:0)</th>
<th>I/O Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>000300-00030f</td>
</tr>
<tr>
<td>01</td>
<td>000310-00031f</td>
</tr>
<tr>
<td>10</td>
<td>000320-00032f</td>
</tr>
<tr>
<td>11</td>
<td>000330-00033f</td>
</tr>
</tbody>
</table>

Table A.1: I/O Address

The current design connects both **SW1** and **SW0** to ground.

The output signals are described next.

**RESETn:** is converted from **RESET** to provide a power-on reset to bring the MIP to a known state before its operation.

**CS(3:0):** selects the corresponding Memory Control chip.

**WR.ENn:** connects the data path between the PC bus and the input ports of all memories.

**RD.ENn:** connects the data path between the output ports of all memories and the PC bus.

**PROC.RWn:** connects the data path between the input port of selected memory and the output port (**Y.out**) of ALU2.
PC.WEn: generates the write enable signal for a Memory Controller when the data transfer from the host computer to the specified memory is requested.

REGSn(13:0): are the register control signals used by the Master Controller, the ALU1, the MAP, the ALU2, and the Volume Adder.

REGENW: updates the register specified by the I/O address.

REGENR: enables the host computer to read the register specified by the I/O address.

REGENRn: is the invert of the output signal REGENR. This signal is not used in the VHDL model.

W.CLK: generates the write enable signal for load the mask.

MEMCS16n: indicates the 16-bit data transfer on the present bus cycle.

The VLSI version of the Bus Interface does not exist. Chris Insalaco decided to use the Bus Interface in FPGA version instead. In the next section, the VHDL model of the circuit will be discussed.

A.2 VHDL Model of the Bus Interface

The schematics of the Bus Interface is depicted in figure A.1. The VHDL model is based on this circuit and the BLM model of the old design. There are seven processes in the model. Processes bus_la and bus_sd produces intermediate signals, while processes bus_reset, bus_bale, bus_iown, bus_memwn, and pc_wen generate the output signals based on the intermediate and input signals.

Process bus_la essentially models the Memory Decoder in the Bus Interface. The signal validmem indicates whether the address presented on PC BUS is valid for the current cycle. The signal balmem is used to activate MEMCS16n.

```vhdl
bus_la:
PROCESS
VARIABLE la_dummy: qsim_state_vector(4 DOWNTO 0);
VARIABLE la_bus: integer := UNKNOWN;
VARIABLE baseok: integer; -- equivalent to sd(6)
BEGIN
WAIT ON la, addrctrl, bale;
```
baseok := (addrctrl/64) MOD 2;
la_dummy := la;
la_bus := in_gen(la_dummy);
IF ((addrctrl MOD 64) = la_bus) AND (baseok = 1) THEN
  validmem <= '1';
  IF bale = '1' THEN
    balmem <= '0';
    validmem_not <= '0'; -- output of U7
  END IF;
ELSE
  validmem <= '0';
  balmem <= '1';
  IF bale = '1' THEN
    validmem_not <= '1'; -- output of U7
  END IF;
END IF;
END PROCESS bus_la;

Process bus_sd generates intermediate signals addrctrl and segment. addrctrl contains the contents of the Address Control Register. Meanwhile, segment contains the contents of the Segment Control Register to control the memory segments. In addition, this process generates the output signals CS(3:0), which is determined by the Segment Control Register. Bit 5 of segment is a flag for mask load or memory access. When the flag is 0, bits 4 and 3 select one of the memory controllers to use the address from the PC BUS for memory access. When the flag is 1, no memory controller is selected and each memory uses the address generated by its own controller.

bus_sd:
PROCESS
  -- VARIABLE DECLARATIONS
BEGIN
  WAIT ON sd, resetn_inter, regenw_inter, regs;
  sd_val := in_gen(sd);
  IF resetn_inter = '0' THEN
    addrctrl <= 0;
    segment <= 0;
  ELSIF regenw_inter'EVENT AND regenw_inter = '1' THEN
    IF regs = 16#7fff# THEN
      addrctrl <= sd_val;
    ELSIF regs = 16#bfff# THEN
      segment_dummy := sd_val;
      segment <= sd_val;
    END IF;
  END IF;
END IF;
index := (segment_dummy/8) MOD 4;
IF (segment_dummy/32 MOD 2) = 0 THEN -- segment(5) = '0'
    CASE index IS
        WHEN 0 => cs_val := 1;
        WHEN 1 => cs_val := 2;
        WHEN 2 => cs_val := 4;
        WHEN 3 => cs_val := 8;
        WHEN OTHERS => cs_val := 0;
    END CASE;
ELSE
    cs_val := 0;
END IF;
out_gen(cs_val, cs_dummy);
cs <= TRANSPORT cs_dummy AFTER CS_DLY;
END PROCESS bus_sd;
END bus_interface_behavior;

Process bus_reset generates the output signal RESETn by inverting the input signal reset.

Process bus_bale produces the output signal MEMCS16n.

Process bus_iow is modeling the component IO_DECODER. The process first decodes the bits SA3 through SA0 to generate register control signals REGSn(15:0); it will then activate REGENR and REGENRn if the host computer issues I/O read. The last output signal processed is REGENW, which is controlled by a state machine.
bus_iown:
PROCESS
-- VARIABLE DECLARATIONS
BEGIN
WAIT ON sa, iorn, iown, resetn_inter, bclk, sw0, sw1;
sa_val := in_gen(sa); -- read PC SA bus
-- evaluate the new regsn signals
index := sa_val MOD 16;
regsn_val := REGS_VALUES(index);
regsn <= regsn_val;
out_gen(regsn_val, regsn_dummy);
FOR i IN regsn'RANGE LOOP
  regsn(i) <= TRANSPORT regsn_dummy(i) AFTER REGSN_DLY;
END LOOP;
regsn_14 <= regsn_dummy(14);
regsn_15 <= regsn_dummy(15);
-- change regenr and regenrn accordingly
IF (sa_val/64 = 16#c#) AND (sw1 = sa(5)) AND (sw0 = sa(4))
  AND iorn = '0' AND iown = '1' THEN
  regenr <= TRANSPORT '1' AFTER REGSN_DLY;
  regenrn <= TRANSPORT '0' AFTER REGSN_DLY;
ELSE
  regenr <= TRANSPORT '0' AFTER REGSN_DLY;
  regenrn <= TRANSPORT '1' AFTER REGSN_DLY;
END IF;
-- construct the regenw state machine
IF resetn_inter = '0' THEN
  regw := 0;
ELSIF bclk'EVENT AND bclk = '1' THEN -- edge triggered on bclk
  IF regw = 0 AND iown = '0' THEN
    regw := 1;
  ELSIF regw = 1 THEN
    regw := 2;
  ELSIF regw = 2 AND iown = '1' THEN
    regw := 0;
  END IF;
END IF;
-- to generate regenw and regenw_inter
IF (sa_val/64 = 16#c#) AND (sw1 = sa(5)) AND (sw0 = sa(4))
  AND regw = 1 AND iorn = '1' THEN
  regenw_dummy := '1';
ELSE
  regenw_dummy := '0';
END IF;
regenw_inter <= regenw_dummy;
regenw <= TRANSPORT regenw_dummy AFTER OUT_DLY;
END PROCESS bus_iown;
The memory I/O, on the other hand, is handled by the process BUS_MEMWn. It is noticed that the signal \textit{PROC.RWn} is generated by inverting the bit 5 of segment register. Since bit 5 is used as flag to indicate the memory or mask load, we will have to load an image to the on-board memory before loading the mask. Otherwise, the resultant image will not be stored in the specified memory. The generation of output signals \textit{RD.ENn}, \textit{WR.ENn}, and \textit{W.CLK} are straight forward.

211 \begin{verbatim}
bus_memwn: PROCEDURE
BEGIN
    WAIT ON memrn, memwn, segment, validmem_not, bclk;
    IF (segment/32 MOD 2) = 1 THEN
        proc_rwn <= TRANSPORT 'O' AFTER OUT_DLY;
    ELSIF (segment/32 MOD 2) = 0 THEN
        proc_rwn <= TRANSPORT '1' AFTER OUT_DLY;
    END IF;
    IF memrn = '0' AND memwn = '1' AND ((segment/32 MOD 2) = 0)
        AND validmem_not = '0' THEN
        rd_enn <= TRANSPORT '0' AFTER OUT_DLY;
    ELSE
        rd_enn <= TRANSPORT '1' AFTER OUT_DLY;
    END IF;
    IF memrn = '1' AND memwn = '0' AND ((segment/32 MOD 2) = 0)
        AND validmem_not = '0' THEN
        wr_enn <= TRANSPORT '0' AFTER OUT_DLY;
    ELSE
        wr_enn <= TRANSPORT '1' AFTER OUT_DLY;
    END IF;
    IF memrn = '1' AND memwn = '0' AND ((segment/32 MOD 2) = 1)
        AND validmem_not = '0' THEN
        w_clk <= TRANSPORT '1' AFTER OUT_DLY;
    ELSE
        w_clk <= TRANSPORT '0' AFTER OUT_DLY;
    END IF;
    -- construct the pc_wen state machine
    IF bclk·EVENT AND bclk = '0' THEN -- negative edge triggered
        IF memrn = '1' AND memwn = '0' AND ((segment/32 MOD 2) = 0)
            AND validmem_not = '0' AND wrenn = 0 THEN
            wrenn <= 1;
        ELSE
            wrenn <= 0;
        END IF;
    END IF;
END PROCEDURE bus_memwn;
\end{verbatim}
The state machine in the above process is used to generate the signal \( PC\_WE_n \), which is simply the inversion of \( WRE_n \).

It should be realized that the current design of the Bus Interface is only for \( 512 \times 512 \) images. Since bit 0 in the Address Control Register is compared with the address bit 19 on the PC bus, the base address has to be reconfigured in software when the address bit 19 is changed for an image of \( 1024 \times 1024 \).
Appendix B

Controller

In the previous chapter, the interface between the host computer and the MIP board was described. We will explore in this chapter the control mechanism of the MIP. There are two types of controllers in the MIP system: the Master Controller and the Memory Controller. The Master Controller is responsible for synchronizing operations between the ALU1, the MAP, the ALU2, and the Volume Adder. The Memory Controller is in charge of assigning memory for a source or a destination image. There are four Memory Controllers, each controls its own memory unit. The following section is devoted to the Master Controller.

B.1 Master Controller

We have mentioned earlier that the Master Controller is used to synchronize operations between the different image processing units, e.g, the ALU1, the MAP, the ALU2, and the Volume Adder. Therefore, the outputs of the Master Controller are connected with all of the image processing units as well as the Memory Controller. Most of the inputs are from the bus interface. The inter-connections between the different circuit component models can be found in figure 5.3. We will discuss in this section the inputs and outputs of the Master Controller, the differences between the FPGA version and VLSI version in circuit design, and the implementation of the circuit modeling in VHDL.

B.1.1 Inputs and Outputs

Most of the input and output descriptions are taken from materials provided by Jens Rodenberg.[10] We will first describe the inputs. Again, it should be realized that a signal
name without the letter "n" at the end indicates that the signal is active high, while a signal with "n" at the end indicates that the signal is active low.

**REGS.START**: selects the start register when **REGENW** is also active. The selected register will start the processor.

**REGS.PI**: selects the processor's instruction/status register when either **REGENW** or **REGENR** is also active. The register contains two write-only instruction bits (bits 0 and 1) and two read-only status bits (bits 6 and 7). Bit 0 determines the operation of MAP being either erosion (bit 0 = 0) or dilation (bit 0 = 1); bit 1 is the bus mode selection which assigns the X2 bus to be either the input of ALU1 (when bit 1 = 0) or the input of ALU2 (when bit 1 = 1); bit 6 is raised high when the processor is ready to accept the next instruction; bit 7 is high when the processor finishes the processing.

**REGS.MS**: selects the memory select register when **REGENW** is also active. The register provides the information on the memory locations of source image and destination image for an image processing. The Master Controller uses this information to route memory control signals to the appropriate Memory Controller, to select the appropriate bus to connect the ALU1, the MAP and the ALU2 with their corresponding memories, and to allow the host computer to read an image from any of the on-board memories. The details were described in Table 4.6 in chapter 4.

**REGENW**: enables the host computer to write to one of the three registers described above.

**REGENR**: enables the host computer to read from the processor's instruction/status register described above.

**SD(7:0)**: is the 8-bit data bus from register read or register write.

**SIZE**: selects the image size to be either 512 x 512 (when **SIZE** = 0) or 1024 x 1024 (when **SIZE** = 1).

**PL.START**: starts the pipelined processor.

**S0.2**: adjusts the number of delays which is determined by the MAP operation.

**CLK**: is the on-board system clock.
CLRn: clears all flip-flops upon a power up of the system.

The output signals are described next.

PL.START.NEXT: generates the start signal for the next pipelined processor.

START.MEM(3:0): issues start signals to the corresponding Memory Controllers specified by the active bits. The Memory Controller will start its address counter from zero upon the next rising clock edge after receiving the start signal.

WRITE.MEM(3:0) instructs the corresponding Memory Controller specified by the active bit that its associated memory will be written to, starting when the START.MEM signal is issued. This will cause the selected memory controller to issue the write enable signal to its memory during valid memory address.

X1.BUS.SELn: connects the memory specified by the active bit to the X1 bus by enabling the buffer between them.

X2.BUS.SELn: connects the memory specified by the active bit to the X2 bus by enabling the buffer between them.

INIT ALU1: instructs the ALU1 to load its next instruction upon the next rising clock edge. INIT ALU1 also initializes the ALU1 that the first valid pixel of an image is its operand upon the next rising clock edge.

MAX: specifies that the operation of the MAP being either erosion or dilation.

START PROC: starts the MAP operation upon the next rising clock edge. The MAP uses this signal to latch the window values and the morphological operation for the next image being processed.

ROWBLNK(5:0): informs the MAP which rows are to be blanked.

COLBLNK(5:0): informs the MAP which columns are to be blanked.

INIT ALU2: instructs the ALU2 and the Volume Adder to load their next instructions upon the next rising clock edge and resets the ALU2 and the Volume Adder.

STOP ALU2: informs the ALU2 and the Volume Adder that their operands upon the next rising clock edge will no longer be a valid pixel for the current operation.
The input and output signals described above are based on the VLSI version. The FPGA version does not include input signals $SIZE$, $PL_{-}START$, and $SO_{-}2$; it does not generate output signal $PL_{-}START_{-}NEXT$. The detailed differences between the two versions are described in the next subsection.

### B.1.2 VLSI Version vs. FPGA Version

There are two differences in terms of the functionality of the Master Controller between the two versions. One is that the VLSI version is capable of generating the control signals for either $512 \times 512$ images or $1024 \times 1024$ images, while the FPGA version is fixed for $512 \times 512$ images. The actual size of the image in the VLSI version is determined by the jumper connecting the input $SIZE$ to either $VCC$ or $ground$. When $SIZE$ is connected to $ground$, the image is $512 \times 512$; when $SIZE$ is connected to $VCC$, the image is $1024 \times 1024$. The other difference is that the VLSI version is designed to adopt to pipelined operations: it accepts $PL_{-}START$ to start the processor and generates $PL_{-}START_{-}NEXT$ for the next processor.

Other differences between the two versions in circuit design exist as well. Although the differences do not affect the functionality of the Master Controller, they do affect implementation of the VHDL model. The first difference occurs at the start register which is a D flip-flop. In the FPGA version, the start signal at the output of the start register stays high for only one clock period regardless of the duration of its input signal. In the VLSI version, however, the start signal at the output of the register stays high as long as the input is high. The second difference is in the number of the delay stages, which is determined by the MAP. In the FPGA version, it requires 15 clock cycles (or stages) for the target pixel to go through the MAP and enter the ALU2. In the VLSI version, it requires only 8 stages. The input $SO_{-}2$ in the VLSI version is used to adjust the number of delay stages. The third difference is related to the Start Blank and Blank Counter blocks and will be discussed in VHDL modeling of that circuit in the next subsection. The schematics of the VLSI and FPGA versions are shown in figure B.1 and B.2.

### B.1.3 VHDL Model of the Master Controller

The VHDL model for the Master Controller is based on the BLM model of the FPGA version and extended to adopt the new features in the VLSI version. The Master Controller
Figure B.1: Schematic of Master Controller in the VLSI version
Figure B.2: Schematic of Master Controller in the FPGA version
in either version can be categorized into three stages according to the signals’ flow: the input stage, the clocked stage, and the output stage. The first stage is the input stage. It accepts the input signals from the input ports and from the clocked stage, then generates the output signals to the other two stages as well as to the output ports. The second stage is the clocked stage. It accepts the signals from the input stage and input ports, and generates the output signals for the input stage, output stage, and output ports. The input CLK is only used in the second stage to synchronize the signals. The third stage is the output stage, which accepts signal from the other two stages as well as from the input ports, and generates the signals for the output ports. The inter-connections between the stages are shown in figure B.3. In the VHDL model, only these interconnections are defined as signals to connect the different processes. This is to minimize the use of the signals.

Each of the stages is modeled with one or more processes. We will discuss each process according to the stages classified above.

**Input Stage**

The input stage consists of four processes: sd_handler, switch, ctrl_reg, and gen_start. Sd_handler is used to handle the bidirectional data bus. Since the INOUT port type was not implemented in system 1076 version 7.0, a buffer called sd.buffer is used to mimic the bidirectional bus.

```vhdl
149  sd_handler:
150     PROCESS
151     BEGIN
152         WAIT ON sd_in, sd_buffer;
153         IF sd_in’EVENT THEN
154             sd_buffer <= sd_in;
155             sd_out <= "ZZZZZZZZ";
156         ELSIF sd_buffer’EVENT THEN
157             sd_out <= sd_buffer;
158         END IF;
159     END PROCESS sd_handler;
```

The process ctrl_reg is to generate the signals bus_mode, max, and mem_instr. Bus_mode and max have been explained in section B.1.1. Mem_instr configures START_MEM(3:0), WRITE_MEM(3:0), X1_BUS_SELn(3:0), and X2_BUS_SELn(3:0). The details for configuring memories and buses were described in table 4.6.
Figure 7.3 Process Flow of the Master Controller
PROCESS

-- VARIABLE declarations.
BEGIN
WAIT ON regenw, regs_pin, regs_msn, sd_buffer,
clrn, start;
IF clrn = '0' THEN
  bus_mode <= '0';
ELSE
  IF regenw = '1' AND regs_pin = '0' THEN
    max <= TRANSPORT sd_buffer(0) AFTER DELAY_MAX;
    IF start = '1' THEN
      bus_mode <= sd_buffer(1);
    END IF;
  END IF;
  IF (regenw = '1' AND regs_msn = '0') THEN
    mem_instr <= in_gen(sd_buffer);
  END IF;
END IF;
END PROCESS ctrl_reg;

The process switch is separated from the process ctrl_reg since the input signals are
fixed for any MIP board. Simulation would be inefficient if the switch and ctrl_reg were
combined into one process. The input signals for the process are SIZE and S0_2, which were
described in section B.1.1. The output signals are sb_max, bc_max, and s0_2_val. Sb_max
and bc_max are the maximum numbers that Start Blank Counter and Blank Counter will
reach respectively.

PROCESS(size,s0_2)
VARIABLE s0_2_val_dummy: integer;
BEGIN
CASE size IS
  WHEN '0' => sb_max <= 32*3+4+2-1; -- should be 512*3+5
  bc_max <= 31;  -- should be 511
  WHEN '1' => sb_max <= 64*3+4+2-1; -- should be 1024*3+5
  bc_max <= 63;  -- should be 1023
  WHEN OTHERS =>
    ...
END CASE;
s0_2_val_dummy := in_gen(s0_2);
s0_2_val <= s0_2_val_dummy;
IF s0_2_val /= UNKNOWN THEN
  del_length_1 <= 6+s0_2_val_dummy;
ELSE
It should be mentioned that the numbers used in the process for \( sb_{\text{max}} \) and \( bc_{\text{max}} \) given in the listing on 114 are configured to limit the simulation time. The actual numbers for the MIP are in the comment lines. Since both counters start from 0, the corresponding maximum is one less than the size of the counter. The general formula for the blank count maximum is

\[
bc_{\text{max}} = \text{row size} - 1. \quad (B.1)
\]

Blank Counter generates the row and column blanking signals for the MAP. The \textit{done} signal from the Blank Counter indicates to the MAP that the last pixel is its operand.

The general formula for the start blank maximum, however, is different from the VLSI version and the FPGA version. In VLSI version,

\[
sb_{\text{max}} = \frac{N - 1}{2} \times \text{row size} + \frac{N + 1}{2} + 2 - 1. \quad (B.2)
\]

\( N \) is the window size fixed at 7 for a \( 7 \times 7 \) window. Since Start Blank is to synchronize the operations between the ALU1 and the MAP, the number of clock periods between the rising edge of \textit{INIT\_ALU1} and rising edge of \textit{START\_PROC} should be \( \frac{N - 1}{2} \times \text{row size} + \frac{N + 1}{2} \). Two extra clock cycles are needed to account for the delay in the operation of ALU1. Referring to figure B.1, it is noticed that \textit{INIT\_ALU1} is generated one clock cycle after the \textit{start} signal of the Start Blank Counter. In addition, one more clock cycle is needed in the ALU1 to perform either addition or comparison before the target pixel is sent to the MAP.

In FPGA version,

\[
sb_{\text{max}} = \frac{N - 1}{2} \times \text{row size} + \frac{N + 1}{2} + 2 - 1 - 1. \quad (B.3)
\]

The difference of one clock cycle between the two versions is due to different designs for the blank counter. Combinational logic is used in the VLSI version, while a state machine is used in the FPGA version. The result is that the \textit{row blank} signals and \textit{column blank} signals in the FPGA version are activated one clock after the \textit{start} signal of the Blank Counter is activated. On the other hand, the blank signals in the VLSI version are activated in the same clock cycle as the \textit{start} signal of Blank Counter is activated. Therefore, a delay is
inserted after the Blank Counter’s start signal in the FPGA version to generate start_proc, which should be synchronized with the first row/column blank signals. To account for the inserted delay, the counter in Start Blank is one less than the required number.

The process gen_start is to generate the start signal for the MIP operation. It accepts the signals start_buf from the clocked stage and from the input port PL_START, and generates the output signal start.

```vhdl
192  gen_start:
193    PROCESS
194    BEGIN
195      WAIT ON pl_start, start_buf;
196      start <= start_buf OR pl_start;
197    END PROCESS gen_start;
```

Clocked Stage

There is only one process, clocked_blocks, in this stage. This process is the core of the Master Controller since it deals with all the synchronized signals for the ALU1, the MAP, the ALU2 and the Volume Adder operations. It generates pro_rdy, pro_done, x1_en, x2_en, start_x1, start_x2, and star_y for the output stage; it generates signals for the output ports INIT_ALU1, INIT_ALU2, START_PROC, STOP_PROC, ROWBLNKs, COLBLNKs, and PL_START_NEXT; it also generates signal start_buf which in turn generates the start signal for the MIP. Since the output ports have been described in section 7.1.1, we will explain here only the output signals which are not connected with the output ports.

*Pro_rdy:* is the input signal of the output stage to indicate that the processor is ready to accept the next instruction. The signal is the bit 6 of the instruction/status register.

*Pro_done:* is the input signal of the output stage to indicate that the processor finishes the processing. The signal is the bit 7 of the instruction/status register.

*X1_en:* enables the first latch in BUS_SELECT to generate X1_BUS_SELn.

*X2_en:* enables the second latch in BUS_SELECT to generate X2_BUS_SELn.

*Start_x1, start_x2, start_y:* are the input signals of the output stage to generate output ports START_MEM(3:0) and WRITE_MEM(3:0).
Start_buf: is the output of the start register. It is the input signal of the process gen_start in input stage.

Before the code for process is presented, we will briefly describe the related part of the circuit in figure B.1 to understand the basic functions being modeled. Upon receiving the start signal, START.BLANK starts to count from 0. When sb_max is reached, it generates START.PROC. Meanwhile, the same signal is entered into BLANK.COUNTER to initialize the counter. As we described in section 7.1.1, START.PROC indicates that the target pixel in the MAP is the first pixel in an image. The various delay stages are required for the MAP to perform the operation. After the pixel is processed, it enters the ALU2. This event is signaled by the INIT.ALU2. STOP.ALU2 is active at bc_max + 1 clock cycles after the INIT.ALU2 being active. In addition, start_x1 is the same signal as start, but start_x2 depends on the bus mode, which was described in section 7.1.1.

The information within the process is passed by using variables. Therefore, it is extremely important to order the statements in the right sequence so that the correct propagation of the information is guaranteed. The rule of the thumb is to assign the value of a variable to the last clock stage first, and the first clock stage last. Since the code is well documented, we will not explain the code in more details.

226          clocked_blocks:
227          PROCESS
228          ... VARIABLE declarations.
240          BEGIN
241          WAIT ON clk, clrn;
242          IF clrn = '0' THEN
261          ... Initialization of the registers.
261          ELSIF clk'EVENT AND clk = '1' THEN
262          ----To generate pro_rdy and pro_done signals----
263          IF start = '1' THEN
264          pro_rdy <= '0';
265          END IF;
266          ELSE
267          IF inter_init_alu2 = '1' THEN
268          pro_rdy <= '1';
269          END IF;
270          IF inter_stop_alu2 = '1' THEN
271          pro_done <= '1';
272          END IF;
273          END IF;
To generate \( x_1 \text{en}, \ x_2 \text{en}, \) and \( \text{start} \_y \) output signals.

\[
dummy \_\text{start} := ((\text{NOT reg} \_\text{startn}) \ AND \ \text{regenw}) \ OR \ \text{pl} \_\text{start};
\]

\[
\text{start} \_x1 <= \ dummy \_\text{start};
\]

\[
x_1 \_\text{en} <= \ \text{start};
\]

\[
x_2 \_\text{en} <= \ \text{start} \_x2;
\]

\[
\text{start} \_y <= \ \text{inter} \_\text{init} \_\text{alu2};
\]

----To propagate the internal signals of the blocks through

----the variable assignment. The order in assigning the

----value is extremely important: the assignment should

----start from the last stage in the chain.

\[
\text{inter} \_\text{pl} \_\text{start} \_\text{next} := \ \text{inter} \_\text{init} \_\text{alu2};
\]

\[
\text{inter} \_\text{init} \_\text{alu1} := \ \text{start};
\]

\[
\text{inter} \_\text{init} \_\text{alu2} := \ \text{del1} \_\text{last};
\]

\[
\text{inter} \_\text{stop} \_\text{alu2} := \ \text{del2} \_\text{last};
\]

FOR \( i \) IN \( \text{MAX} \_\text{DEL} \_1 \) DOWNTO 1 LOOP

\[
\text{del1}(i) := \ \text{del1}(i-1);
\]

\[
\text{del2}(i) := \ \text{del2}(i-1);
\]

END loop;

\[
\text{del1}(0) := \ \text{sb} \_\text{done};
\]

\[
\text{del2}(0) := \ \text{bc} \_\text{done};
\]

CASE \( s0 \_2 \_\text{val} \) IS

\[
\text{WHEN} \ 0 => \ \text{del1} \_\text{last} := \ \text{del1}(6);
\]

\[
\text{del2} \_\text{last} := \ \text{del2}(6);
\]

\[
\text{WHEN} \ 7 => \ \text{del1} \_\text{last} := \ \text{del1}(13);
\]

\[
\text{del2} \_\text{last} := \ \text{del2}(13);
\]

\[
\text{WHEN} \ \text{OTHERS} => \null;
\]

END CASE;

\[
\text{start} \_x2 \text{ is level sensitive to its inputs. Therefore,}
\]

\[
\text{the assignment is placed after the assignment of its}
\]

\[
\text{input variables.}
\]

IF \( \text{bus} \_\text{mode} = '1' \) THEN

\[
\text{start} \_x2 <= \ \text{del1} \_\text{last};
\]

ELSE

\[
\text{start} \_x2 <= \ \text{dummy} \_\text{start};
\]

END IF;

\[
\text{Process the blank counter next}
\]

IF \( \text{sb} \_\text{done} = '1' \) THEN

\[
\text{rowb} \_\text{cnt} \_\text{dummy} := \ 0;
\]

\[
\text{colb} \_\text{cnt} \_\text{dummy} := \ 0;
\]

\[
\text{bc} \_\text{done} := '0';
\]

ELSIF \( \text{rowb} \_\text{cnt} \_\text{dummy} = \ \text{bc} \_\text{max} \) AND \( \text{colb} \_\text{cnt} \_\text{dummy} = \ \text{bc} \_\text{max} \) THEN

\[
\text{rowb} \_\text{cnt} \_\text{dummy} := \ 0;
\]

\[
\text{colb} \_\text{cnt} \_\text{dummy} := \ 0;
\]

ELSIF \( \text{colb} \_\text{cnt} \_\text{dummy} = \ \text{bc} \_\text{max} \) THEN
colb_cnt_dummy := 0;
rowb_cnt_dummy := rowb_cnt_dummy+1;
ELSE
    colb_cnt_dummy := colb_cnt_dummy+1;
END IF;
IF rowb_cnt_dummy = bc_max AND colb_cnt_dummy = bc_max THEN
    bc_done := '1';
ELSE
    bc_done := '0';
END IF;

---- To process the start blank counter. This part has to be
---- placed after blank counter since the variable sb_done
---- used in blank counter is generated here.
IF start = '1' THEN
    sb_cnt := 0;
    sb_done := '0';
ELSIF sb_done = '1' THEN
    sb_done := '0';
ELSIF sb_cnt = sb_max-1 THEN
    sb_done := '1';
    sb_cnt := sb_cnt+1;
ELSE
    sb_cnt := sb_cnt+1;
END IF;

---- To process the regs_startn signal
IF regenw = '1' AND regs_startn = '0' THEN
    start_buf <= '1';
ELSE
    start_buf <= '0';
END IF;
END IF;

---- To generate the row and column blanks for their output ports.
IF rowb_cnt_dummy = 0 THEN
    out_gen(16#38#,rowblnk_temp);
ELSIF rowb_cnt_dummy = 1 THEN
    out_gen(16#30#,rowblnk_temp);
ELSIF rowb_cnt_dummy = 2 THEN
    out_gen(16#20#,rowblnk_temp);
ELSIF rowb_cnt_dummy = (bc_max-2) THEN
    out_gen(16#01#,rowblnk_temp);
ELSIF rowb_cnt_dummy = (bc_max-1) THEN
    out_gen(16#03#,rowblnk_temp);
ELSIF rowb_cnt_dummy = bc_max THEN
    out_gen(16#07#,rowblnk_temp);
ELSIF rowb_cnt_dummy = bc_max THEN
    out_gen(16#07#,rowblnk_temp);
ELSE
... The generation of column blank is identical as above.

Output Stage

There are two processes in this stage. The process Mem_bus_reg is to model BUS_SELECT and MEM>Select in figure B.1. $X_1.BUS\_SELn(3:0)$ and $X_2.BUS\_SELn(3:0)$ are generated by BUS_SELECT; $START\_MEM(3:0)$ and $WRITE\_MEM(3:0)$ are generated by MEM>Select. The following code is to model the functionality of BUS_SELECT and MEM>Select. The decoding mechanism on $mem\_instr$ (memory instruction) is based on the integer manipulation rather than the logic manipulation. The functionality to be described is better understood with an integer presentation.

```vhdl
mem_bus_reg:
PROCESS
... VARIABLE declarations
BEGIN
WAIT ON mem_instr, start_x1, start_x2, start_y,
x1_en, x2_en, clrn;

----To generate the value for mem_sel_buffer----
IF clrn = '0' THEN
  mem_sel_buffer := 0;
ELSIF start_x1 = '1' THEN
  mem_sel_buffer := mem_instr;
END IF;

----To generate the value for writing memory----
w_mem_val := 0;
FOR i IN 0 TO MEM_SEL_INDEX LOOP
  IF extract_bits(mem_sel_buffer,2*i+1,2*i) = 2 THEN
    w_mem_val := w_mem_val+2**i;
  END IF;
```
END LOOP;
out_gen(w_mem_val,temp_write_mem);

----To generate the value for starting memory----
FOR i IN 0 TO MEM_SEL_INDEX LOOP
  CASE extract_bits(mem_sel_buffer,2*i+1,2*i) IS
    WHEN 0 => temp := start_x1;
    WHEN 1 => temp := start_x2;
    WHEN 2 => temp := start_y;
    WHEN OTHERS => temp := '0';
  END CASE;
  IF temp = '0' THEN
    temp_value := 0;
  ELSIF temp = '1' THEN
    temp_value := 1;
  END IF;
  CASE i IS
    WHEN 0 => start_mem_val := temp_value;
    WHEN 1 => start_mem_val := 2*temp_value +start_mem_val;
    WHEN 2 => start_mem_val := 4*temp_value +start_mem_val;
    WHEN 3 => start_mem_val := 8*temp_value +start_mem_val;
    WHEN OTHERS => null;
  END CASE;
END LOOP;
out_gen(start_mem_val,temp_start_mem);

----To generate the value for selecting x1 bus----
IF (extract_bits(mem_instr,7,4) = 0) THEN
  CASE extract_bits(mem_instr,3,0) IS
    WHEN 0 => x1_buf_val := 16#0f;
    WHEN 1 => x1_buf_val := 16#0e;
    WHEN 2 => x1_buf_val := 16#0d;
    WHEN 4 => x1_buf_val := 16#0b;
    WHEN 8 => x1_buf_val := 16#07;
    WHEN OTHERS => x1_buf_val := 16#0f;
    ASSERT (FALSE)
    REPORT "Bus conflict in hardware! No bus selected."
    SEVERITY ERROR;
  END CASE;
  IF clrn = '0' THEN
    x1_buf_val := 16#0f;
  END IF;
  ELSIF clrn = '0' THEN
    x1_buf_val := 16#0f;
  ELSIF x1_en = '1' THEN
x1_buf_val := 0;
FOR i in 0 TO MEM_SEL_INDEX LOOP
  IF (extract_bits(mem_instr,2*i+1,2*i) /= 0) THEN
    x1_buf_val := x1_buf_val+2**i;
  END IF;
END LOOP;
END IF;
out_gen(x1_buf_val, x1_bus_dummy);

----To generate the value for selecting x2 bus-----
IF clrn = '0' THEN
  x2_buf_val := 16#0f#;
ELSIF x2_en = '1' THEN
  x2_buf_val := 0;
  FOR i in 0 TO MEM_SEL_INDEX LOOP
    IF (extract_bits(mem_instr,2*i+1,2*i) /= 1) THEN
      x2_buf_val := x2_buf_val+2**i;
    END IF;
  END LOOP;
END IF;
END IF;
out_gen(x2_buf_val, x2_bus_dummy);

----Output the signals to the corresponding ports-----
write_mem <= TRANSPORT temp_write_mem AFTER DELAY_WRT_STR_MEM;
start_mem <= TRANSPORT temp_start_mem AFTER DELAY_WRT_STR_MEM;
x1_bus_seln <= TRANSPORT x1_bus_dummy AFTER DELAY_X_BUS;
x2_bus_seln <= TRANSPORT x2_bus_dummy AFTER DELAY_X_BUS;
END PROCESS mem_bus_reg;

The reading of instruction/status register is modeled by status_reg, which is listed below. In the process, the variable regrst is used as a switch between status register and sd_buffer. The switch is turned on whenever the register is addressed, and the switch is turned off whenever the register is de-addressed. If the switch is at off position, the value in sd_buffer will not be affected by the instruction/status register. The implementation of the switch is based on the function of read enable for instruction/status register in actual hardware. It should be realized that the sd_buffer is driven by two sources in the same architecture: one from status_reg and one from sd_handler. The code is written in the way that only one source is turned on at any time. If the code needs to be updated to system 1076 version 8.0, a resolution function should be written for this multi-source driven signal. In system 1076 version 7.0, however, the resolution function is not implemented, and the value on the multi-driven signal is overwritten by the newest signal value.
B.2 Memory Controller

A Memory Controller is to assign its corresponding memory to a source image or a destination image by providing the address and the write enable signals for the memory. We will describe in the following subsections the inputs and outputs of the Memory Controller and the corresponding VHDL model.

B.2.1 Inputs and Outputs

The number of inputs for the Memory Controller is small. \( CLK \) is the on-board system clock, while \( CLK_2 \) is a delayed signal of \( CLK \) and is used to generate proper memory write enable pulses. \( Size \) indicates the size of the image. \( START\_MEM(3:0) \) and \( WRITE\_MEM(3:0) \) are the outputs of the Master Controller and have been described in section 7.1.1. \( CLRn \) reset the memory counter and write enable register. \( PC\_CS \) and \( PC\_RD \) are used to generated output signals and are described below.

There are only two outputs from each Memory Controller: \( MEM\_ADDR(19:0) \) and \( WEn \). \( MEM\_ADDR(19:0) \) is the address line for the corresponding memory, and \( WEn \) is the active-low write enable signal. The origin of these signals could be from either the host computer or the Memory Controller, depending on the status of \( PC\_CS \). When \( PC\_CS \) is low, \( MEM\_ADDR(19:0) \) and \( WEn \) are generated by the Memory Controller itself; when \( PC\_CS \) is high, \( MEM\_ADDR(19:0) \) is connected with \( PC\_ADDR(19:0) \), and \( WEn \) is connected with \( PC\_RD \). Therefore, the host computer is granted the access to one of the memories if the corresponding \( PC\_CS \) is high.

B.2.2 VHDL Model of the Memory Controller

The schematic of a Memory Controller chip is shown in figure B.4.

From figure B.4, it is seen that the Memory Controller is mainly consisted of a memory counter, three multiplexors, and a register. The memory counter is started from 0 when \( START\_MEM \) is high, or \( CLRn \) is low, or the number reaches the highest address. The function is accomplished by the process mem_counter.

```vhdl
mem_counter:
PROCESS
VARIABLE count : integer := 0;
BEGIN
```
Figure B.4: Schematic of Memory Controller
WAIT ON clk UNTIL clk = '1';
IF (start_mem = '1' OR clrn = '0' OR count = maxcnt) THEN
  count := 0;
  done <= '0';
ELSIF start_mem = '0' THEN
  count := count+1;
  IF count = maxcnt THEN
    done <= '1';
  END IF;
END IF;
addr <= count;
END PROCESS mem_counter;

The status of the PC_CS determines the source of the corresponding memory address and the write enable signal. This is implemented by the processes source_of_mem_addr and generate_wen respectively.

source_of_mem_addr:
PROCESS
  VARIABLE temp_addr : integer;
  VARIABLE mem_delay: time := 0 ns;
BEGIN
  WAIT ON pc_cs, addr, new_pc_addr;
  IF pc_cs = '0' THEN
    temp_addr := addr;
    mem_delay := 25 ns;
  ELSIF pc_cs = '1' THEN
    temp_addr := new_pc_addr;
    mem_delay := 40 ns;
  ELSE
    temp_addr := UNKNOWN; -- This result is not specified in BLM
  END IF;
  new_mem_addr <= TRANSPORT temp_addr AFTER mem_delay;
END PROCESS source_of_mem_addr;

Addr is the signal generated by the process mem_counter, new_pc_addr is the address from the host computer, and new_mem_addr is the address for the next memory read/write cycle.

generate_wen:
PROCESS
BEGIN
  WAIT ON clk2, pc_cs, we, pc_rd;
  IF pc_cs = '0' THEN
IF clk2='1' AND we='1' THEN
    wen <= TRANSPORT '0' AFTER 30 ns;
ELSE
    wen <= TRANSPORT '1' AFTER 30 ns;
END IF;
ELSIF pc_cs = '1' THEN
    wen <= TRANSPORT pc_rd AFTER 50 ns;
END IF;
END PROCESS generate_wen;

The signal we is the output of the write enable register and is generated by the process generate_we.

generate_we:
PROCESS
BEGIN
    WAIT ON clk UNTIL clk = '1';
    IF clrn = '0' THEN
        we <= '0';
    ELSIF clrn = '1' THEN
        IF start_mem = '0' THEN
            IF done = '1' THEN
                we <= '0';
            END IF;
        END IF;
        ELSIF start_mem = '1' THEN
            we <= write_mem;
        END IF;
    END IF;
END PROCESS generate_we;
Appendix C

Utilities

The following utilities were used during this project. The general procedures of creating an ASCII format image file were:

- Scan a photo or graph by the Xerox 7650 Scanner. Output of the scanner is an image file in TIFF format.
- Convert the TIFF format into IMG format. The IMG format can be used for frame-grabber and the MIP board.
- Convert the IMG format into ASCII format. The VHDL model reads the ASCII format.
- Convert the ASCII format into PostScript format. The PostScript format can be used for HP LaserJet Printer.

C.1 XEROX 7650 Scanner

The scanner is connected with the PC, "Beaker." Do not turn on the scanner until you start the scanner program. Type "scan7650" after the DOS prompt to start the scanner application. Choose the TIFF file format which (may be) is the only working format.

C.2 TIFF to IMG

Use the program, TIFTOIMG under the directory CONVERT on BEAKER.
C.3 IMG to ASCII / ASCII to IMG and Display an IMG Image

Use the program, IMAGEIO under the directory WEICHUN on Gateway 2000 486/25c.

C.4 ASCII to PS

Use the program, PS under the directory WEICHUN on Gateway 2000 486/25c. For the 512/time512 images, copy the c program to APOLLO workstations and recompile the program.

C.5 Display a PostScript Image on PC

Use the program, GS, under the directory GS. GS is the GostScript which is much more powerful than display a PS file on screen. Read the documentation.

C.6 Connect PC with Apollo Workstations - DPCI

On the Gateway 2000 486/25c, type "start." After the prompt shown, type in your Apollo username and password to login. The D drive is your Apollo account. The lpt2: is connected to the printer lj for ASCII file. The lpt3: is connected to the printer server fj for POSTSCRIPT file. Read the DPCI menu before use it. Before you leave PC, type "stop" to disconnect the link.

C.7 Print out a PostScript on LaserJet on Apollo

Use the following commands: prf -pr fj -trans FILENAME